An educational tool for visualising actor programs

Andela Ćolak*, Marin Aglić Ćuvić*
* Faculty of Science, University of Split, Croatia
acolak@pmfst.hr, maragl@pmfst.hr

Abstract - Object-oriented programming (OOP) is probably the most popular programming paradigm today. Because of this, it is often taught at faculties associated with computer science. The general idea of OOP is to break a complex problem into abstractions called objects which interact with each other via method invocation.

On the other hand, the actor model provides a computation model based on agents that are called actors. Actors in the actor model are similar to objects in OOP, but unlike objects are concurrent and communicate by message passing.

At our faculty, a game-based approach is used to teach students about OOP. The actors in a game can be taught as agents that interact with an environment. Furthermore, students program agent behaviour in our artificial intelligence course. Therefore, students have experience with both objects and agents.

However, students still find it difficult to understand the actor model and build applications with it. In order to help students, we have developed a visualisation system for actor-based applications. We collected responses on the system after presenting it to students that were taught the model. In this paper, we i) present the developed visualisation system and ii) report the results on the responses collected.

Keywords – actor model; message-passing; notional machine; object-oriented programming; program visualisation

I. INTRODUCTION

Computer science students at the Faculty of Science learn about the actor model as part of their graduate study. We chose this concurrent computational model because it most closely resembles object-oriented programming (OOP) [1]. The actor model has been used for building multiplayer games (with virtual actors, but based on the actor model) [2], Ericsson’s 3G mobile network [3], Intel’s big data streaming engine [4], and other applications that require a high degree of scalability [5, 6].

In literature, the actor model was first introduced as a formalism in artificial intelligence [7]. The basis of this model is an actor, that is an agent that is invoked by receiving messages. The basic premise is that actors can uniformly model any mode of behaviour, or rather that any mode of behaviour can be defined as sending messages to actors.

Whereas actors are the basis of the actor model, objects are the basis of OOP [1]. However, the main difference between an actor and an object is that the former is concurrent. Therefore, actors execute their actions concurrently and asynchronously with other actors. The actor model is even sometimes referred to as a combination of an object-based view of the world with concurrency. All computer science related undergraduate study programmes at the Faculty of Science in Split include a course in OOP.

Therefore, we were surprised when many students, despite its similarities to OOP, found programming with the actor model very difficult. One of the main issues seemed to be that students failed to understand how to organise the communication between actors, even for basic examples with none or minimal concurrency. It seemed as students failed to see the connection between sending messages and receiving responses and invoking methods and receiving a return value. Because of this, we believe that students were missing an essential perspective on OO programs - as a collection of interacting objects.

Another issue for students may have been the inability to access an actors’ state directly. However, students were taught about and programmed agent-based models in NetLogo [8]. NetLogo agents, similar to actors, cannot directly access the state of other agents. Instead, they need to ask one another to do something, which we may think about as message passing.

In order to write programs and reason about them, students need a valid model of the notional machine. The notional machine is an abstraction that results from the constructs of the programming language and their relationship to the to the execution of the program [9]. There are different notional machines for different programming languages and paradigms [10]. Even a single language can have more than one notional machine.

Sorva suggests that understanding OOP requires two notional machines [10]. The first one is an extension of a notional machine for imperative programming, while the second one describes the communication between objects. These provide different perspectives on OOP. With regards to the difficulties described earlier, students might have experienced OOP as merely an extension of imperative programming. Therefore, they might be
missing the other perspective on object-oriented programs.

In order to address the mentioned issues, we have developed an educational tool for visualising a high-level notional machine that focuses on the interchange of messages. The tool provides students with a graph-based layout of the exchanged messages and a timeline that shows when each of them was sent. We presented the tool to a group of students and collected their responses via a questionnaire. In this paper, i) we present the visualisation tool that we developed at the Faculty of Science in Split and ii) report on the student responses regarding the visualisation presented to them.

II. PROGRAM VISUALISATIONS AND NOTIONAL MACHINES

Program visualisation tools have often been developed in order to visualise the notional machine with the aim to provide students with a correct model of the machine [11]. Over as many as fifty different visualisation tools have been identified in the literature [11, 12]. Therefore, it has become relatively difficult to come up with new and unique ways of visualising execution. Tools such as JaguarCode have started including different the visualisation of different aspects of code to complement one another [13].

The visualisation that we introduce here is most similar to Novis [14], akka-viz [15] and akka-visualmailbox [16]. Novis visualises the execution of OO programs with a focus on communication. It also provides a view at different levels of detail. However, it is not intended for the actor model, and it currently is unknown to us if it is actively developed or abandoned.

Akka-viz [15] and akka-visualmailbox [16] are open source systems found on Github which visualise actors and their message exchange. However, neither appears to be an educational tool. Akka-visual-mailbox visualises the number of messages exchanged between actors, but that seems to be all. Akka-viz seems to provide more details but is developed to be a debugging tool and not an educational one. Both systems are developed for use with Akka and are therefore unusable with .NET, which our students use. Akka-visualmailbox visualises message exchange by collecting sent messages using a custom type of actor mailbox. We use the same approach for collecting messages in our tool. On the other hand, Akka-viz uses AspectJ for instrumentalisation. Furthermore, given that no significant updates have been logged in years, it would seem that both systems have been abandoned.

The goal of the tool we present here is to facilitate students learning about the actor model. It provides an additional view of a timeline that none of the other presented systems has, can also be used with OO programs and new features are currently in development.

III. ACTOR MODEL

The actor model is a mathematical model for concurrent computations [17]. We already mentioned some of its applications in the introduction. It is often used to build applications that need to handle tasks asynchronously, such as parallel programming, programming of distributed systems, distributed artificial intelligence [18] and many more. Although our students have seen agents in AI when programming agent-based models, the actor model gives them an asynchronous note.

The reason for its broad application in the field of computer science lies in the fact that the actor model alleviates the developers from having to take care of thread management and locking [19]. This level of abstraction enables the usage of multi-threaded behaviour without the use of low-level concurrency constructs [20].

The basis of the actor model are actors, which are computational agents that communicate via message passing. When compared to software agents, actors are passive and react only when a message is received. Upon receiving a message, an actor may send messages to other actors, create new ones or update its local state. Updating local state may include changing the actors’ behaviour. Any actions the actor takes is concurrent and asynchronous with other actors. Actors cannot access the local state of other actors; they can only influence other actors by sending messages.

Each actor has an associated mailbox which stores the received messages [17]. Messages are stored in order of arrival and processed synchronously in first-in-first-out (FIFO) order. All messages are processed synchronously. Therefore, all behaviour inside a single actor is synchronous, which greatly simplifies reasoning about actors.

Actors are very similar to objects in OOP. Both of these entities encapsulate some state and manifest behaviour [1]. Invoking a method on an object is even sometimes referred to as message passing. Unlike actors, objects can directly access and modify each other’s state. However, we can limit the accessibility of an objects’ state by using properties and the appropriate access modifiers. By changing some local state, objects may also change their behaviour. Although, it is worth to mention that this is not the same as changing the behaviour of an actor.

The actor model, just like OOP, allows us to decompose programs into a set of self-contained interactive components.

IV. AKKA.NET

Akka.NET is an implementation of the actor model for the .NET framework [20]. It is a port of the popular Akka toolkit for Java and Scala that allows programming message-driven, concurrent and parallel applications [21]. We chose to use Akka.NET in teaching because our students mostly use C# in their courses and because of Akka’s popularity.

V. AKKAVISUAL

Students often find it difficult to understand and program in the actor model. One of the main issues seems to be that students find actor communication too abstract. They often find it difficult to perceive which actors will communicate with each other during program execution and which messages they will exchange. Therefore, we developed AkkaVisual, a web-based tool for visualising
message passing between actors. The tool can also be adapted to visualise the interactions between different objects in OOP. Thereby, for both the actor model and OOP, the tool visualises the higher-level message-passing notional machine.

The tool is a web application that exposes a web API to collect data. The API receives the details about the exchanged messages. Typically, these details will include the sender, receiver, message content and type of message. Once received, the data is pushed to the web client in real-time using SignalR. On the web client, the visualisation is drawn in real-time as the program executes. We use Vis.js to draw the visualisation [22].

Data can be sent to AkkaVisual's exposed web API via a standard HTTP POST request. However, for convenience, we have developed a library that contains a custom mailbox which may be used by Akka.NET actors. To use this mailbox, students need to include it in their project and specify its use when configuring their actor system. The library also keeps track of the order of messages via a vector clock.

Typically, when an actor receives a message, it is placed on the queue in the actor's mailbox. The actor processes the messages in FIFO order. In addition to this, our custom mailbox forwards the details about received messages to AkkaVisual's web API. These details also include the value of the vector clock for the message.

AkkaVisual offers:

- View of messages and actors exchanging those messages
- Information about the type of actor
- Information about the properties of each message
- Slow replay option
- Zoom in and zoom out options and ability to draw graph in the window
- Timeline view
- View of programs running on computers connected to a cluster, containing a legend of which process sent which message

Since AkkaVisual is an open-source project, additional functionalities can be added to an existing library or library can be translated to Scala or Java by anyone in the community. Figure 1 shows an overview of how the system works along with our custom mailbox.

VI. METHODOLOGY

A. Participants and tool presentation

We presented AkkaVisual to a group of students enrolled in second-year graduate studies. Students that did not learn the actor model as part of their graduate curriculum, as well as those that knew who made the tool, were excluded from the study. The latter were excluded to avoid peer bias. We refer to the remaining nineteen students as participants. All of them learned about the actor model as part of their graduate study curriculum during the second semester. However, for reasons that we will discuss later, only eleven participants were considered for the qualitative analysis.

Due to classroom restrictions, the students were not able to try out the tool themselves. Therefore, we introduced the tool via a presentation that included images of the user interface and available features. Additionally, a live demo was carried out. All of this lasted for about 20 minutes, after which the participants had about 10 minutes to fill out a questionnaire.

B. Data collection tool

Data was collected using a questionnaire that consisted of seven 5-point Likert-scale questions, one dichotomous question and a single open question. The Likert-scale questions consisted of one question regarding the tools appropriateness for learning the actor model and the other six regarding the usefulness of existing features and the potential value of planned ones [23]. The purpose of the single open-ended question was to collect any suggestions and complaints the participants might have about the tool presented. The dichotomous question only asked if a legend should be present on the tool's user interface.

C. Data analysis

We analysed the collected quantitative and qualitative data separately.

Cronbach's alpha for the quantitative data is 0.869, which is considered highly reliable [24]. For each item, we report the frequency and distribution of values. Additionally, for the top four features that got the top average scores, we try to give a possible explanation of why the feature got its score.

For the qualitative data, we use content analysis. Weber suggests that content analysis may be used to code open-ended questions [25]. Although the participants' answers are short, they provide useful insight into the students' impression of the usefulness of the tool for learning the actor model. We take an inductive content analysis approach in which we derive the codes and categories from the data [26]. After familiarising ourselves with the data, we assigned the initial codes to segments of the answers. We refined these codes as we reread the data. Once we were satisfied with the codes, we grouped them into categories and counted their frequencies. We report the categories and codes, as well as the frequency of their occurrence.
Out of the total of nineteen students, fifteen left any responses; the others left the question blank. Out of these, two participants left an answer with no information relevant to the study, e.g. “no complaints”. Two others simply stated “great tool”. Therefore, we coded the answers of the remaining eleven participants.

When determining the codes, it is worth noting that multiple codes may have been assigned to the same answer, and also the same code may have been assigned to answers from multiple participants.

VII. RESULTS AND DISCUSSION

The goal of the questions regarding the tool’s features is to obtain feedback on which features participants found most useful and guide future development of new ones. The descriptive statistics are reported in Table 1. The first question is not so much about the usefulness of the tools’ features but rather to see if students feel the tool is appropriate for use by graduate level students. Questions two to five refer to existing features of the tool, and questions six and seven refer to features we plan to add in the future. We can see that the mean scores of all of the questions are above 4. Therefore, we conclude that students believe that all of the existing features and all of the features we plan to add are could be useful for them to learn the actor model.

For brevity, we will further look into and expand only on the top four highest scored features. The top scored question is number 6, which refers to saving and replaying a visualisation without executing the program again (avg. = 4.53). We offer two possible explanations as to why this feature got the top score. The first possibility is that students would like to share the visualisations they obtain without sharing their programs. This would allow students that have completed their assignments to share how their program executes without sending the program itself. Additionally, it would allow students who were not able to complete their assignments to compare the execution of their program with that of their colleagues.

The second explanation is related to the time it takes for the program to run entirely. Students probably do not want to re-run the application over and over again, especially if the computations take a long time or a large number of values need to be input manually.

The second to top scored question is related to the usefulness of displaying the type of actor and details related to the sent message (avg = 4.32). Actor applications often have different types of actors that can handle the same type of message. Hence, we assume that students feel it would help clarify why some behaviour occurred.

The third and fourth top scored questions have the same mean score (avg = 4.32). One of them refers to the usefulness of visualising the concurrency in the system with a timeline. Displaying the exchanged messages in a graph-based layout does not allow simple tracing of the order in which messages were exchanged between actors. Therefore, the timeline provides one way to compensate for this as it shows the exact time each message was sent.

Finally, the last of the top four scored questions refers to sending messages to the actor system from the web client. This feature would mostly be useful for debugging as users could see if the actor can receive certain types of messages and respond to them appropriately.

Regarding the open-ended question, we derived eight codes from the answers of the thirteen participants. We grouped them into three categories: *positive about the tool, the tool needs improvements and tool isn’t useful*. The category positive about the tool contains codes that refer to the participants’ perception that the tool could be useful for learning the actor model. The second category contains the codes related the improvements that need to be made the tool. Finally, the category tool isn’t useful contains codes for reasons that may impede the tool’s usefulness. Figure 2 shows a map of these categories and code. Code frequencies are reported in Table 2.

### Table 1. Frequency scores for the tools’ features

<table>
<thead>
<tr>
<th>#</th>
<th>Question</th>
<th>Score</th>
<th>Mean</th>
<th>Median</th>
<th>Mode</th>
<th>Std. Deviation</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The tool is adapted for first-year graduate students learning about the actor model for the first time.</td>
<td></td>
<td>4.05</td>
<td>4</td>
<td>4</td>
<td>0.911</td>
<td>19</td>
</tr>
<tr>
<td>2</td>
<td>I would like to see a timeline of the execution of each thread or process where the exact time of sending and receiving each message is displayed.</td>
<td></td>
<td>4.21</td>
<td>5</td>
<td>5</td>
<td>0.976</td>
<td>19</td>
</tr>
<tr>
<td>3</td>
<td>The replay feature seems useful.</td>
<td></td>
<td>4.16</td>
<td>4</td>
<td>5</td>
<td>0.898</td>
<td>19</td>
</tr>
<tr>
<td>4</td>
<td>The timeline seems to be useful in visualising the concurrency of the system.</td>
<td></td>
<td>4.32</td>
<td>4</td>
<td>5</td>
<td>0.749</td>
<td>19</td>
</tr>
<tr>
<td>5</td>
<td>It is useful to see the features of the messages sent and the type of the actor.</td>
<td></td>
<td>4.37</td>
<td>5</td>
<td>5</td>
<td>0.761</td>
<td>19</td>
</tr>
<tr>
<td>6</td>
<td>It would be useful to save the generated visualisation and play it without rerunning the program.</td>
<td></td>
<td>4.53</td>
<td>5</td>
<td>5</td>
<td>0.697</td>
<td>19</td>
</tr>
<tr>
<td>7</td>
<td>It would be useful to send a message to an actor from the web application.</td>
<td></td>
<td>4.32</td>
<td>4</td>
<td>4</td>
<td>0.582</td>
<td>19</td>
</tr>
</tbody>
</table>

Figure 2. A map of categories and codes derived by content analysis
Concerning the reported code frequencies, we can conclude that the participants mostly believe the tool could be useful for learning the actor model. In fact, the top three most frequently occurring codes are grouped into the positive about tool category.

Participants have most often expressed that they think the tool will facilitate learning (45,45%) about the actor model in general. The fact that many participants point out this belief suggests that they are aware that students have difficulties to program with actors.

A few participants have expressed their desire to use the tool themselves or in class (27,27%). One of them stated that they think they "will use it [AkkaVisual] when taking the course". Another believes that the tool would have helped him/her achieve a better grade: "I think my results in the course would have been better with the use of this visualisation". Therefore, we may conclude that students are prepared to use the tool in class and while learning.

A couple of participants recognised the difficulty in correctly perceiving actor programs (18,18%). Students may find it hard to trace the execution in actor systems because they find the actor model abstract. One of the participants noted that he would try to sketch actor systems while learning, but would often make mistakes. Participants also noted that the tool could allow theory consolidation (9,09%).

The second category includes codes that do not dismiss the tool but point out that it needs improvements. A few participants noted that the tool requires improved user-friendliness (18,18%) and additional features (18,18%). Participants believe that further work is required on the tool with features mentioned in the questionnaire. Some participants noted that the visualisation is drawn too fast, which, we assume, refers to the replay feature. Slowing down the real-time visualisation during program execution may not be beneficial without slowing down the execution of the program itself. Otherwise, the program would complete much before the visualisation finishes.

Finally, the third category includes those codes that seem to dismiss the usefulness of the tool. One such code is included: "requires theoretical knowledge" (9,09%). One individual believes that having specific theoretical knowledge is required to use the tool. However, the example he provides in tightly related to the example provided in class. Furthermore, a small theoretical background is always provided for whatever is taught, be it OOP, the actor model, or something else.

VIII. LIMITATIONS OF THE STUDY

The main limitation of the study is that the participants did not have a chance to try out the tool themselves and no standardised test was used to score the tool. However, we will address this in a future study.

Therefore, we plan to do a second study in which students will be able to try the visualisation first hand and fill out the system usability scale (SUS) [27]. Furthermore, we plan to conduct a pilot study that will evaluate the tool’s effectiveness in helping students learn about the actor model.

Another limitation of the study is the small number of participants. We retained nineteen participants for the quantitative analysis and eleven for the qualitative. Given that his is a preliminary study and the small number of students at the graduate level, we couldn’t avoid this. To accumulate more participants and confirm our findings, we plan to repeat this study for the next few years.

IX. CONCLUSION

The actor model remains a difficult computational model for students to understand and use to write programs. To facilitate student learning and success in programming with the actor model, we have developed and introduced a visualisation tool called AkkaVisual. Although it still needs some improvements, participants noted that they would like to use the tool. Some of them believe that the tool might have helped them achieve better results when they were learning about the actor model. One of the reasons is that the participants are aware of the difficulties they had when they first started learning the actor model.

Some stated that they had trouble with perceiving actor systems and made mistakes when trying to sketch them on paper. They note that this tool could provide students with a correct perception of actor systems. These findings indicate that we were correct in our assumption that students find actor communication too abstract. However, further research is needed to test this assumption.

Some additional work on the system is required to improve user-friendliness. Some participants believe that additional features should be added to increase its usefulness. The scores for existing and requested features is reported in Table 1. It would seem that most of all participants would like to be able to save and replay a visualisation.

We will continue our work by implementing the requested features based on the provided survey and participant responses. Furthermore, we will give students a chance to try out the tool and introduce a standardised test to score the tool’s usability. Finally, we plan to carry out a study of the tool’s effectiveness in learning the actor model.

X. REFERENCES


“Akka-viz.”


