Development of Alloy interactive teaching materials

Formal Methods for Software Engineering WS22/23

Pramoch Viriyathomrongul, 124244
Harshadeep Gelli, 124303

Bauhaus-Universität Weimar

Abstract. Alloy is a powerful formal specification language and analysis tool that describes and analyzes the structure and behavior of various systems, including software, hardware, networks, and organizations. It uses a combination of first-order logic, set theory, and relational algebra to define models and analyze them for inconsistencies, counterexamples, and other properties of interest.
We aim to develop an interactive application for teaching Alloy language which will serve as a quick entry point to the Alloy world. The application provides language fundamentals in the form of a unit lesson with examples, practice exercises, and informative feedback. With this application, learners will gain a solid understanding of Alloys key concepts and learners can easily pick up where they left off and learn at their own pace.

1 Introduction

1.1 Background

Alloy is a formal specification language and analysis tool that has gained popularity due to its ability to describe and analyze complex systems such as software, hardware, networks, and organizations. It provides a powerful means to specify the structure and behavior of systems in a precise and rigorous manner. By using a combination of first-order logic, set theory, and relational algebra, Alloy enables the creation of models that can be analyzed for inconsistencies, counterexamples, and other properties of interest.

In software engineering, the use of formal methods such as Alloy is essential for ensuring that software systems behave as intended and meet their specified requirements. Without formal design, it can be challenging to express what the system needs to do, leading to potential errors and issues during development. By using Alloy, engineers can verify and validate their designs and ensure that they are building what they need to build.

1.2 Motivation

Despite all the positive things, we need to admit that Alloy has a steep learning curve and is hard for beginner user to start with. There are a lot of relation theories, a lot of operators and a lot of syntax to deal with. Existing resources may be too technical or require prior knowledge of formal methods, making it difficult for beginners to get started with the language.

To address this issue, we aim to develop an interactive console application for teaching Alloy language which will serve as a quick entry point to the Alloy world. The application provides language fundamentals in the form of a unit lesson with examples, practice exercises, and informative feedback. With this console application, learners will gain a solid understanding of Alloys key concepts and learners can easily pick up where they left off and learn at their own pace.

2 Challenge Aspect

This project presents several challenges that need to be addressed in order to develop an effective interactive teaching application for the Alloy language.

2.1 The first challenge is to create efficient lessons and exercises that cover all the fundamental knowledge of the Alloy language

To overcome this challenge, we plan to break down each topic into unit lessons and start with the basic concepts, such as signatures and relationships. As learners progress, we will gradually introduce more complex topics, including predicates, facts, operators (set operators, relation operators, and boolean operators), and quantifiers. Each lesson will also include specific tasks that allow learners to test their understanding and receive feedback on their input.

2.2 Another challenge is to provide learners with informative feedback and solutions as they work on each task

To accomplish this, we will use the Alloy API[1] library to verify user input and provide feedback. Specifically, when a user submits their Alloy solution as an .als file, the application will invoke the Alloy API to perform several checks and analyses. These include a type and syntax check, analysis of dead and core signatures within the input model, and instance information display. The application will also use Alloy GUI Tool to visualize instance information as a graphical graph. Finally, the application will save each instance's information as an .xml file so that learners can come back and run the application with a specific command to visualize the saved .xml file.

[1] For Alloy 6, see https://alloytools.org/

2.3 Performance is a crucial consideration when verifying correctness against the Alloy Tool

Therefore, we will need to ensure that our console application runs at a reasonable performance level. To address this challenge, we will implement performance measurements for critical processes, including Alloy model parsing, Alloy model analyzing, and Alloy command execution. The application will report the measurement information to the user at the same time that the Alloy results are displayed.

2.4 Organize and implement the code in a way that allows for easy addition or expansion of new Alloy lessons in the future

without having to change the entire code base. To accomplish this, we will follow best practices for software development, including modular design and separation of concerns. This will enable us to add new functionality and features without disrupting existing code.

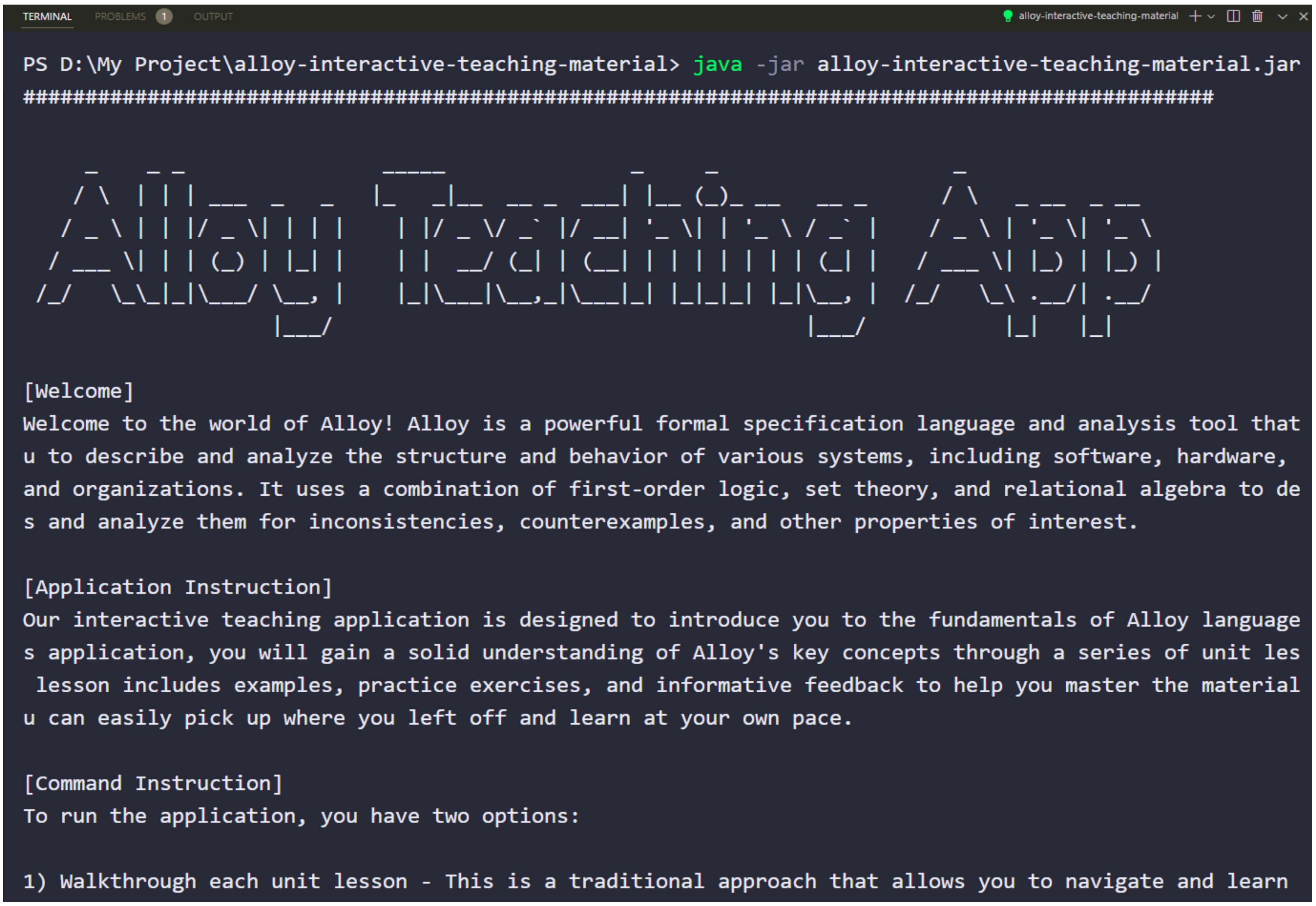
3 Example & Intuition

In this section, we provide an overview of our Alloy teaching application's main features and flow. Our application is divided into two main flows: the lesson flow and the Alloy analyzer flow.

3.1 Lesson Flow

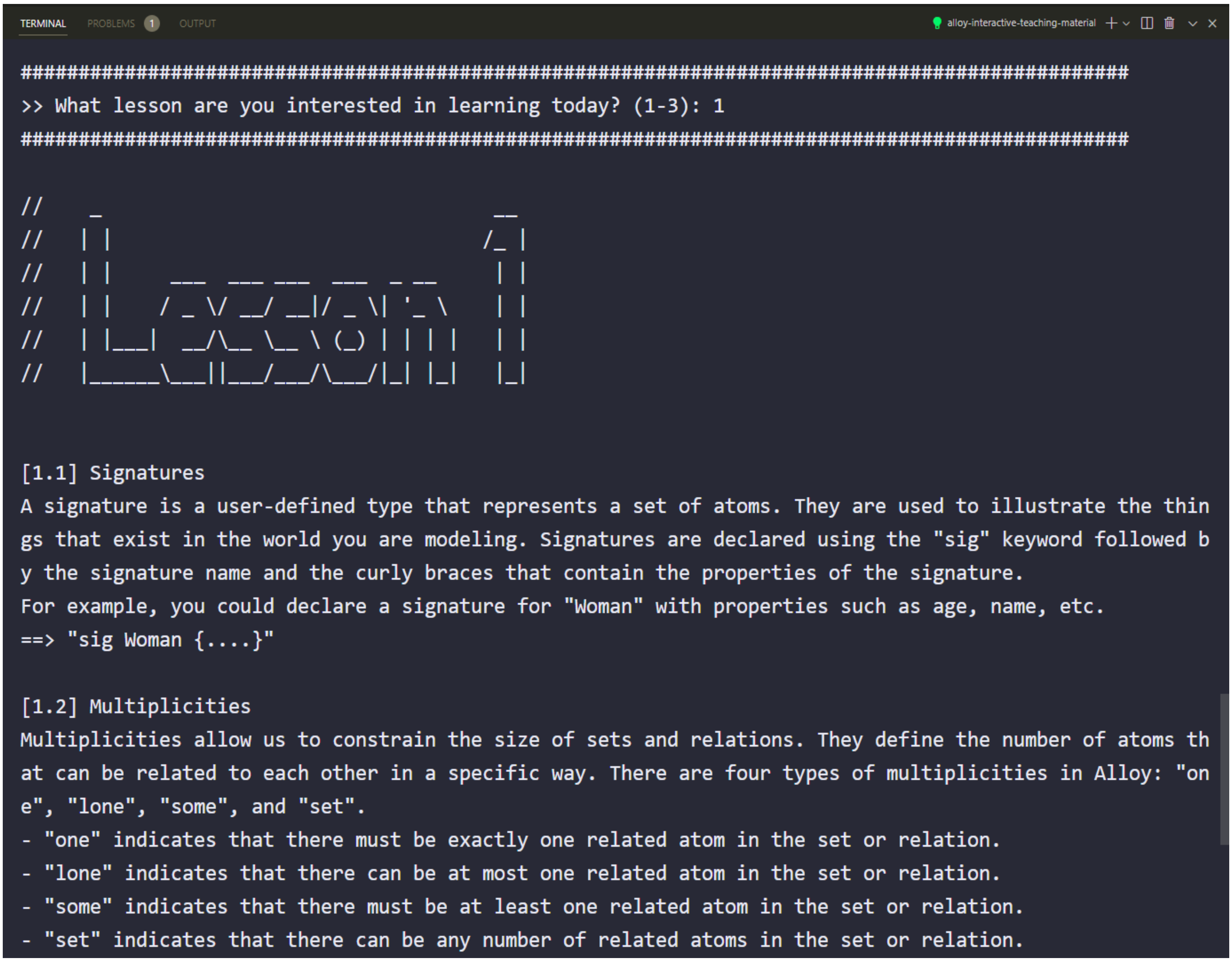
The lesson flow allows users to navigate and learn through desired lessons. After completing a lesson, users are presented with a syntax example and an exercise to test their knowledge of the lesson. Each lesson's sections are saved as .txt or .als files in the same directory as the application, so users can revisit these files later. The initial implementation of the Alloy teaching tool covers three unit lessons, each focusing on a specific topic: 1) Signatures, Relations, Multiplicities, 2) Facts, Functions, Predicates, and 3) All types of operators, Quantifiers.

Figures 1-5 illustrate what the lesson flow looks like when using the **"java -jar alloy-interactive-teaching-material.jar"** command.



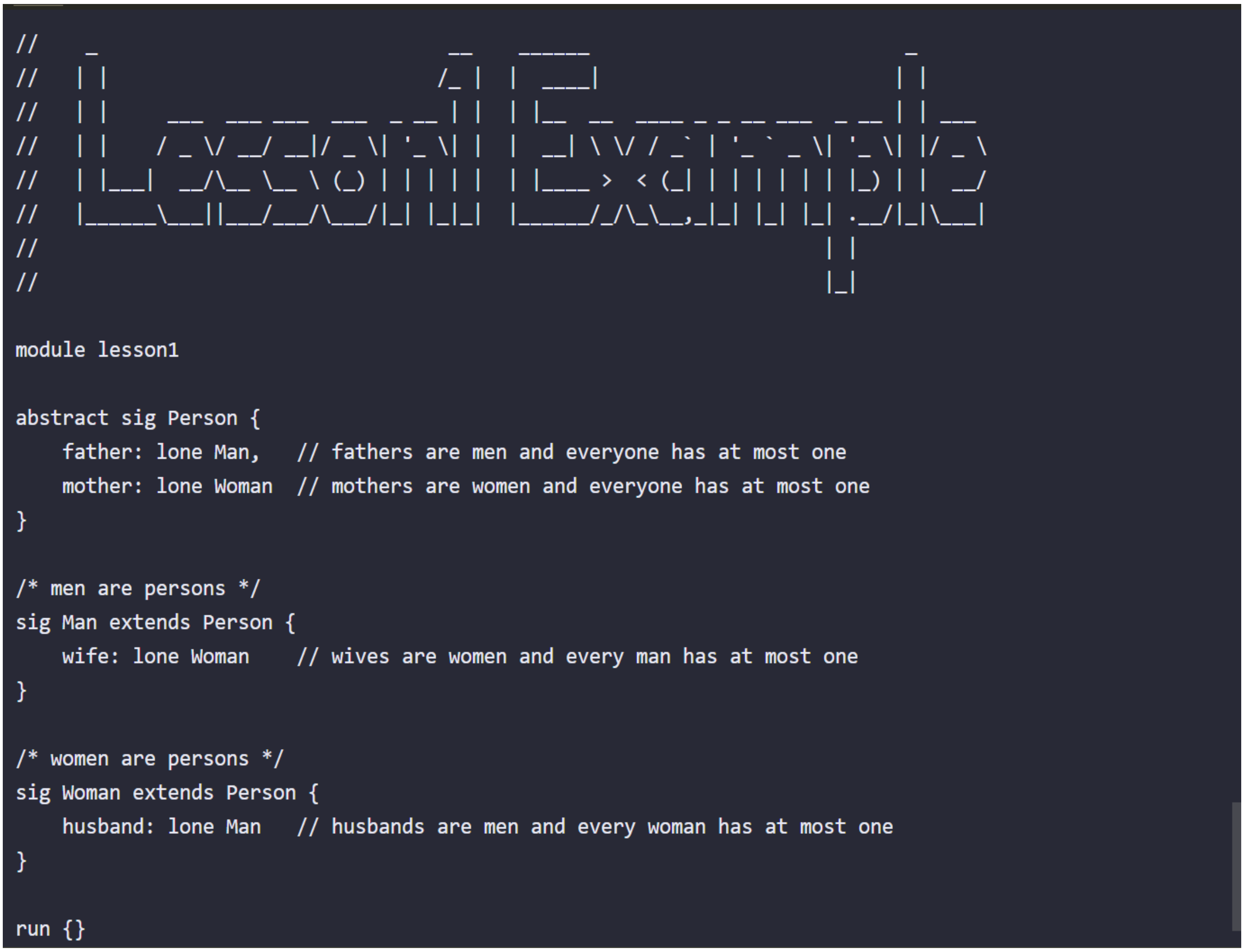
```
PS D:\My Project\alloy-interactive-teaching-material> java -jar alloy-interactive-teaching-material.jar
###############################################################################################


    _    _ _               _____                _     _                  _
   / \  | | | ___  _   _  |_   _|__  __ _  ___| |__ (_)_ __   __ _     / \   _ __  _ __
  / _ \ | | |/ _ \| | | |   | |/ _ \/ _` |/ __| '_ \| | '_ \ / _` |   / _ \ | '_ \| '_ \
 / ___ \| | | (_) | |_| |   | |  __/ (_| | (__| | | | | | | | (_| |  / ___ \| |_) | |_) |
/_/   \_\_|_|\___/ \__, |   |_|\___|\__,_|\___|_| |_|_|_| |_|\__, | /_/   \_\ .__/| .__/
                   |___/                                     |___/          |_|   |_|

[Welcome]
Welcome to the world of Alloy! Alloy is a powerful formal specification language and analysis tool that
u to describe and analyze the structure and behavior of various systems, including software, hardware,
and organizations. It uses a combination of first-order logic, set theory, and relational algebra to de
s and analyze them for inconsistencies, counterexamples, and other properties of interest.

[Application Instruction]
Our interactive teaching application is designed to introduce you to the fundamentals of Alloy language
s application, you will gain a solid understanding of Alloy's key concepts through a series of unit les
 lesson includes examples, practice exercises, and informative feedback to help you master the material
u can easily pick up where you left off and learn at your own pace.

[Command Instruction]
To run the application, you have two options:

1) Walkthrough each unit lesson - This is a traditional approach that allows you to navigate and learn
```

Figure 1 shows the welcome page with instructions on how to navigate throughout the application



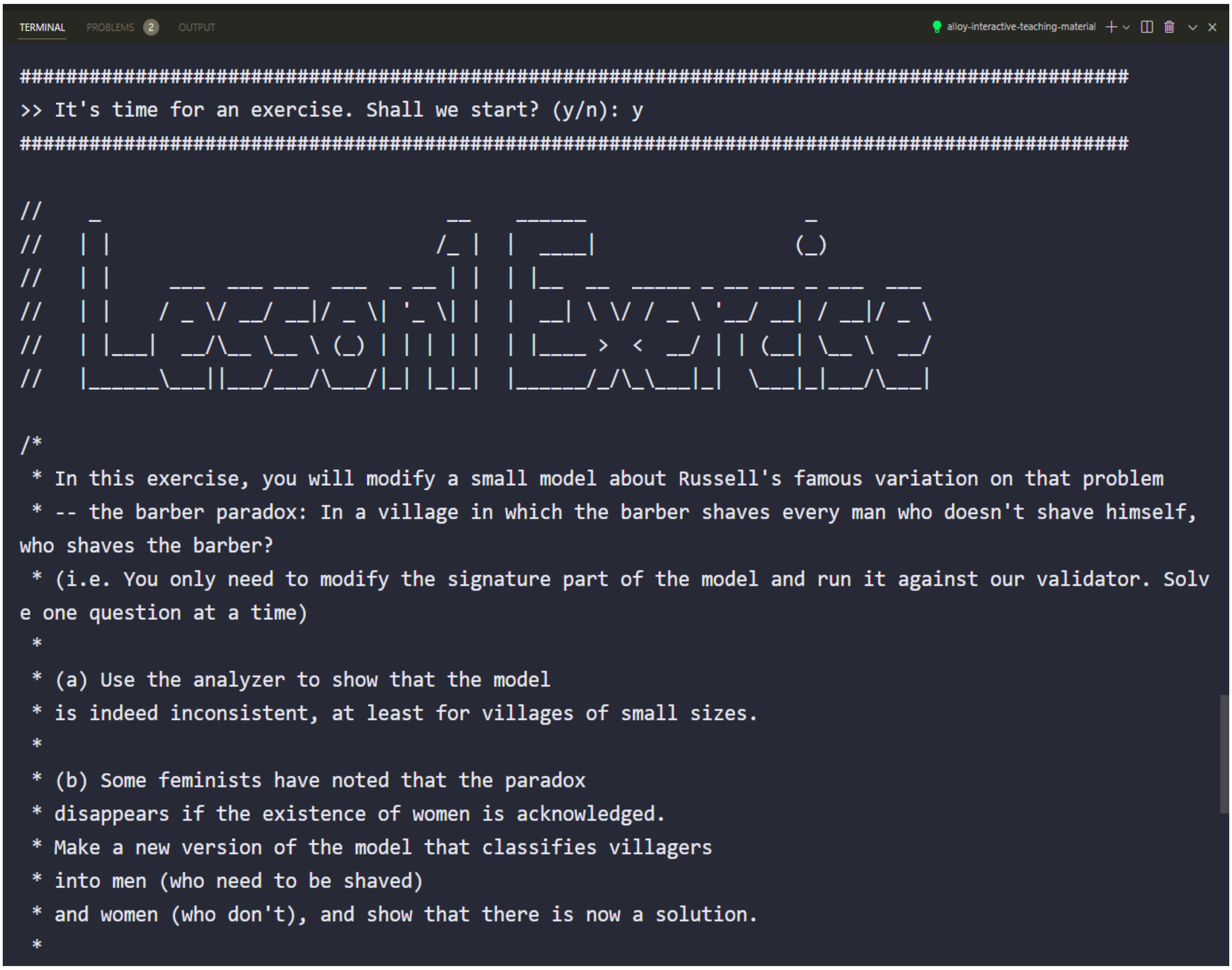
```
################################################################################################
>> What lesson are you interested in learning today? (1-3): 1
################################################################################################


//    _                                  __
//   | |                                /_ |
//   | |     ___  ___ ___  ___  _ __     | |
//   | |    / _ \/ __/ __|/ _ \| '_ \    | |
//   | |___|  __/\__ \__ \ (_) | | | |   | |
//   |______\___||___/___/\___/|_| |_|   |_|



[1.1] Signatures
A signature is a user-defined type that represents a set of atoms. They are used to illustrate the thin
gs that exist in the world you are modeling. Signatures are declared using the "sig" keyword followed b
y the signature name and the curly braces that contain the properties of the signature.
For example, you could declare a signature for "Woman" with properties such as age, name, etc.
==> "sig Woman {....}"

[1.2] Multiplicities
Multiplicities allow us to constrain the size of sets and relations. They define the number of atoms th
at can be related to each other in a specific way. There are four types of multiplicities in Alloy: "on
e", "lone", "some", and "set".
- "one" indicates that there must be exactly one related atom in the set or relation.
- "lone" indicates that there can be at most one related atom in the set or relation.
- "some" indicates that there must be at least one related atom in the set or relation.
- "set" indicates that there can be any number of related atoms in the set or relation.
```

Figure 2 displays the lesson selection screen



```
//    _                                __   ______                           _
//   | |                              /_ | |  ____|                         | |
//   | |     ___  ___ ___  ___  _ __   | | | |__  __  ____ _ _ __ ___  _ __ | | ___
//   | |    / _ \/ __/ __|/ _ \| '_ \  | | |  __| \ \/ / _` | '_ ` _ \| '_ \| |/ _ \
//   | |___|  __/\__ \__ \ (_) | | | | | | | |____ >  < (_| | | | | | | |_) | |  __/
//   |______\___||___/___/\___/|_| |_| |_| |______/_/\_\__,_|_| |_| |_| .__/|_|\___|
//                                                                    | |
//                                                                    |_|

module lesson1

abstract sig Person {
    father: lone Man,   // fathers are men and everyone has at most one
    mother: lone Woman  // mothers are women and everyone has at most one
}

/* men are persons */
sig Man extends Person {
    wife: lone Woman    // wives are women and every man has at most one
}

/* women are persons */
sig Woman extends Person {
    husband: lone Man   // husbands are men and every woman has at most one
}

run {}
```

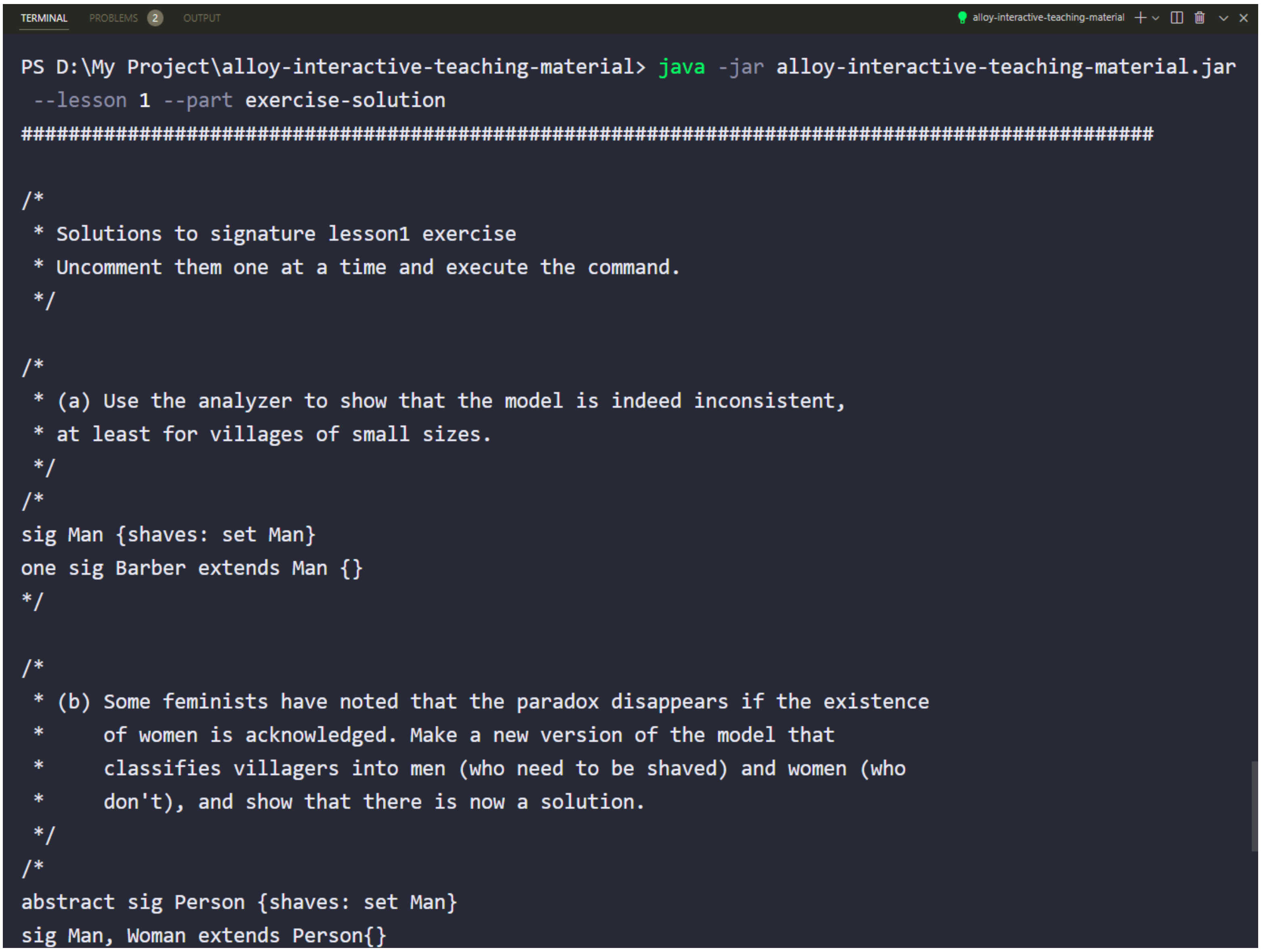
Figure 3 provides a model example for each lesson topic. Each example model is saved as a .als file in the same directory as our application, so users can run a specific command to analyze the model and view an instance later (see section 4.2 for the full command usage)



```
###############################################################################
>> It's time for an exercise. Shall we start? (y/n): y
###############################################################################

//    _                              __   ______                   _
//   | |                            /_ | |  ____|                 (_)
//   | |     ___  ___ ___  ___  _ __ | | | |__  __  _____ _ __ ___ _ ___  ___
//   | |    / _ \/ __/ __|/ _ \| '_ \| | |  __| \ \/ / _ \ '__/ __| / __|/ _ \
//   | |___|  __/\__ \__ \ (_) | | | | | | |____ >  <  __/ | | (__| \__ \  __/
//   |______\___||___/___/\___/|_| |_|_| |______/_/\_\___|_|  \___|_|___/\___|

/*
 * In this exercise, you will modify a small model about Russell's famous variation on that problem
 * -- the barber paradox: In a village in which the barber shaves every man who doesn't shave himself,
who shaves the barber?
 * (i.e. You only need to modify the signature part of the model and run it against our validator. Solv
e one question at a time)
 *
 * (a) Use the analyzer to show that the model
 * is indeed inconsistent, at least for villages of small sizes.
 *
 * (b) Some feminists have noted that the paradox
 * disappears if the existence of women is acknowledged.
 * Make a new version of the model that classifies villagers
 * into men (who need to be shaved)
 * and women (who don't), and show that there is now a solution.
 *
```

Figure 4 displays the specific tasks included in each lesson that allow learners to test their understanding and receive feedback based on their solution model. Users can submit their Alloy model solution using the

"java -jar alloy-interactive-teaching-material.jar --lesson <lesson-number> --part exercise-submit --partParam <solution-file.als>" command.



```
PS D:\My Project\alloy-interactive-teaching-material> java -jar alloy-interactive-teaching-material.jar
 --lesson 1 --part exercise-solution
################################################################################################

/*
 * Solutions to signature lesson1 exercise
 * Uncomment them one at a time and execute the command.
 */

/*
 * (a) Use the analyzer to show that the model is indeed inconsistent,
 * at least for villages of small sizes.
 */
/*
sig Man {shaves: set Man}
one sig Barber extends Man {}
*/

/*
 * (b) Some feminists have noted that the paradox disappears if the existence
 *     of women is acknowledged. Make a new version of the model that
 *     classifies villagers into men (who need to be shaved) and women (who
 *     don't), and show that there is now a solution.
 */
/*
abstract sig Person {shaves: set Man}
sig Man, Woman extends Person{}
```

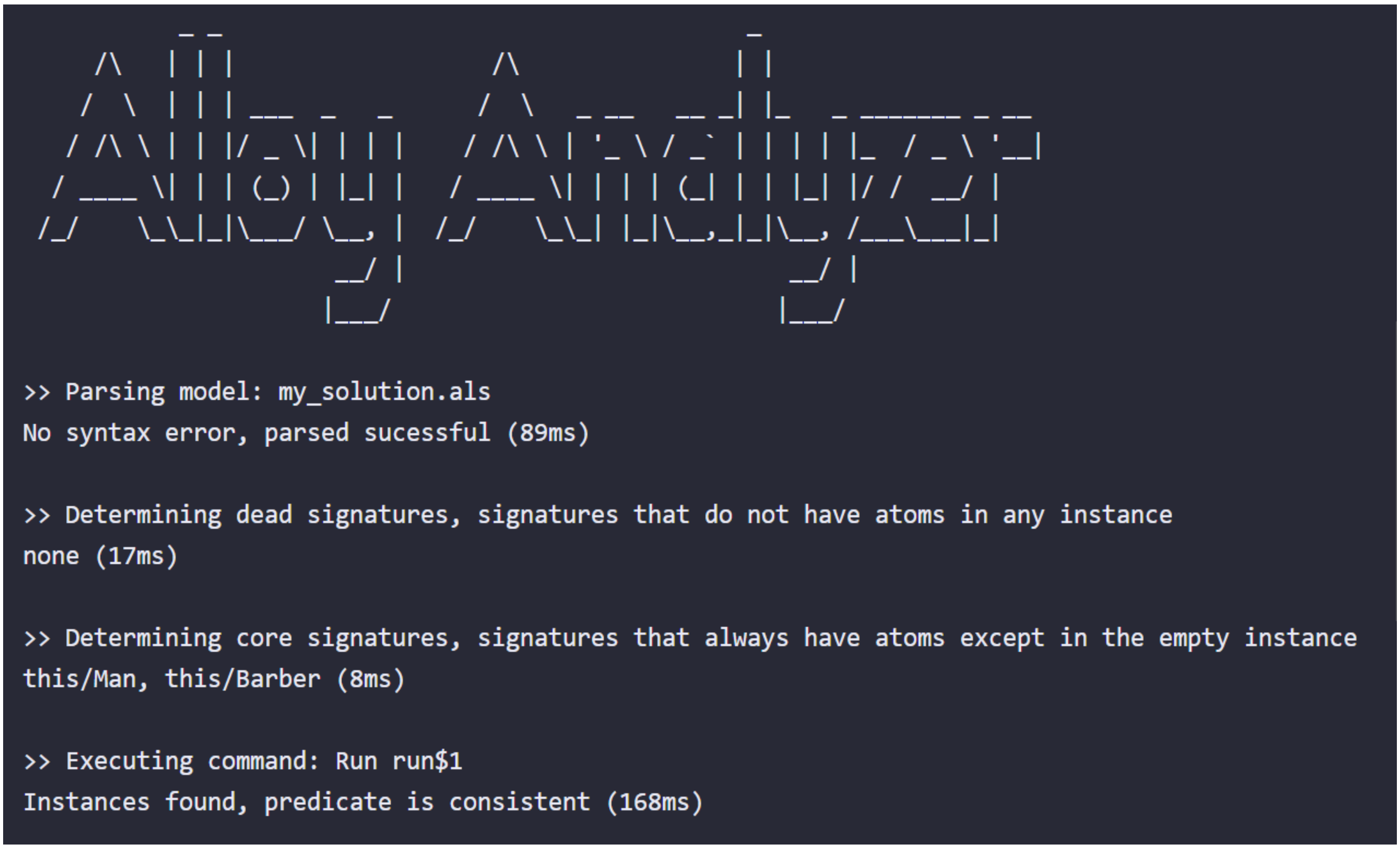
Figure 5 shows the command to view the solution if users struggle with any task

3.2 Alloy Analyzer Flow

The Alloy Analyzer flow is triggered when users submit an Alloy solution file for each exercise. The application invokes the Alloy API to perform several checks and analyze the model, including a type and syntax check, analysis of dead and core signatures, and instance information display. The application also invokes the Alloy API's GUI Tool to visualize instance information as a graphical graph. Finally, the application saves each instance's information as an xml file so that learners can come back and run the application with a specific command to visualize the saved xml file.

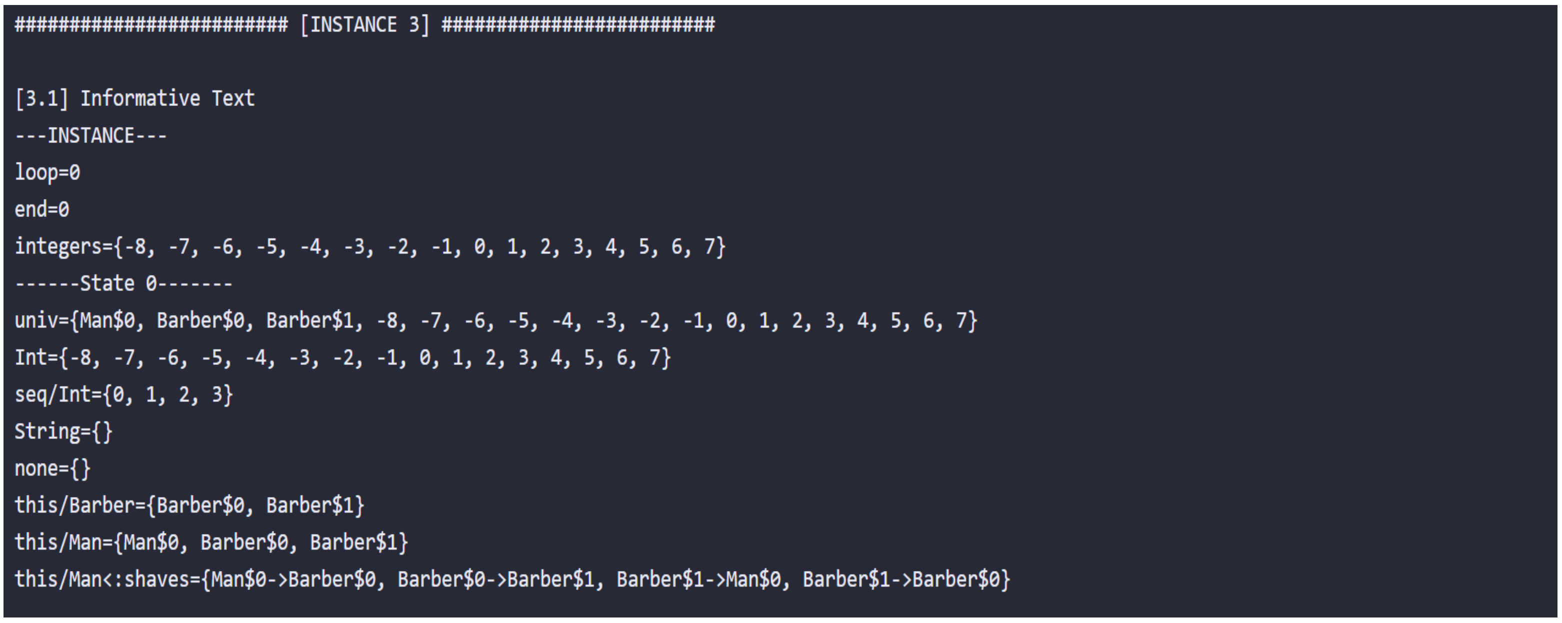
Figures 6-9 illustrate how application display informative feedback from these processes when using the

"java -jar alloy-interactive-teaching-material.jar --lesson 1 --part exercise-submit --partParam my_solution.als" command



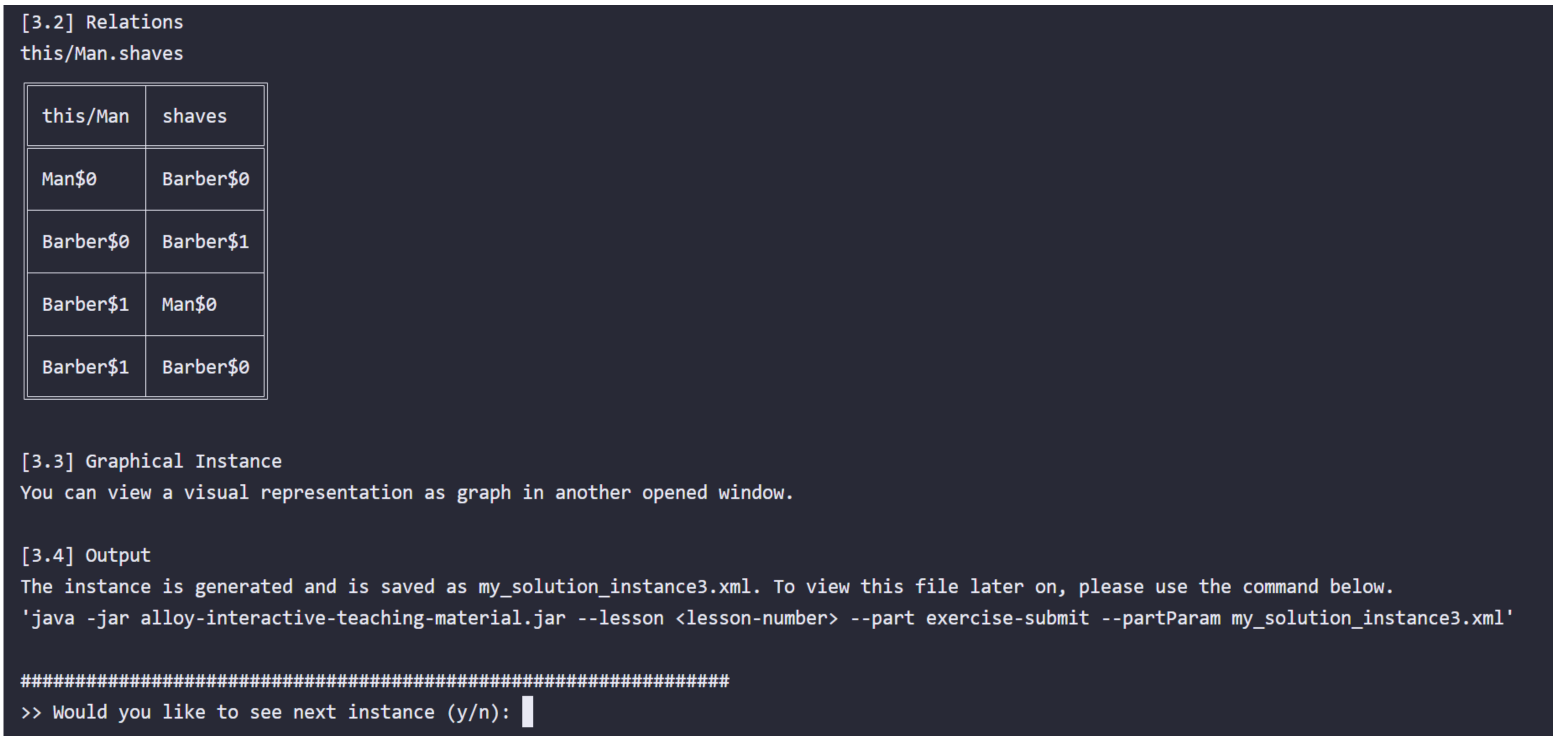
```
          _ _                                     _
    /\   | | |                /\                 | |
   /  \  | | | ___  _   _    /  \   _ __   __ _| |_   _ _______ _ __
  / /\ \ | | |/ _ \| | | |  / /\ \ | '_ \ / _` | | | | |_  / _ \ '__|
 / ____ \| | | (_) | |_| | / ____ \| | | | (_| | | |_| |/ /  __/ |
/_/    \_\_|_|\___/ \__, |/_/    \_\_| |_|\__,_|_|\__, /___\___|_|
                     __/ |                          __/ |
                    |___/                          |___/

>> Parsing model: my_solution.als
No syntax error, parsed sucessful (89ms)

>> Determining dead signatures, signatures that do not have atoms in any instance
none (17ms)

>> Determining core signatures, signatures that always have atoms except in the empty instance
this/Man, this/Barber (8ms)

>> Executing command: Run run$1
Instances found, predicate is consistent (168ms)
```

Figure 6 displays informative feedback to the user, including a syntax check, analysis of dead/core signatures, and results from executing the first command in the my_solution.als model. The application also displays the performance time in each process along with the result.



```
######################## [INSTANCE 3] ########################


[3.1] Informative Text
---INSTANCE---
loop=0
end=0
integers={-8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7}
------State 0-------
univ={Man$0, Barber$0, Barber$1, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7}
Int={-8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7}
seq/Int={0, 1, 2, 3}
String={}
none={}
this/Barber={Barber$0, Barber$1}
this/Man={Man$0, Barber$0, Barber$1}
this/Man<:shaves={Man$0->Barber$0, Barber$0->Barber$1, Barber$1->Man$0, Barber$1->Barber$0}
```

Figure 7 depict an example of the instance information that is displayed to the user



```
[3.2] Relations
this/Man.shaves
```

this/Man	shaves
Man$0	Barber$0
Barber$0	Barber$1
Barber$1	Man$0
Barber$1	Barber$0

```
[3.3] Graphical Instance
You can view a visual representation as graph in another opened window.

[3.4] Output
The instance is generated and is saved as my_solution_instance3.xml. To view this file later on, please use the command below.
'java -jar alloy-interactive-teaching-material.jar --lesson <lesson-number> --part exercise-submit --partParam my_solution_instance3.xml'

###################################################################
>> Would you like to see next instance (y/n):
```

Figure 8 depict an example of the instance information that is displayed to the user

The information in Figure 7 and Figure 8 includes the result text from the Alloy API, a table of signature relations, instance information as a graphical graph (Figure 9), and the saved instance as an xml file.

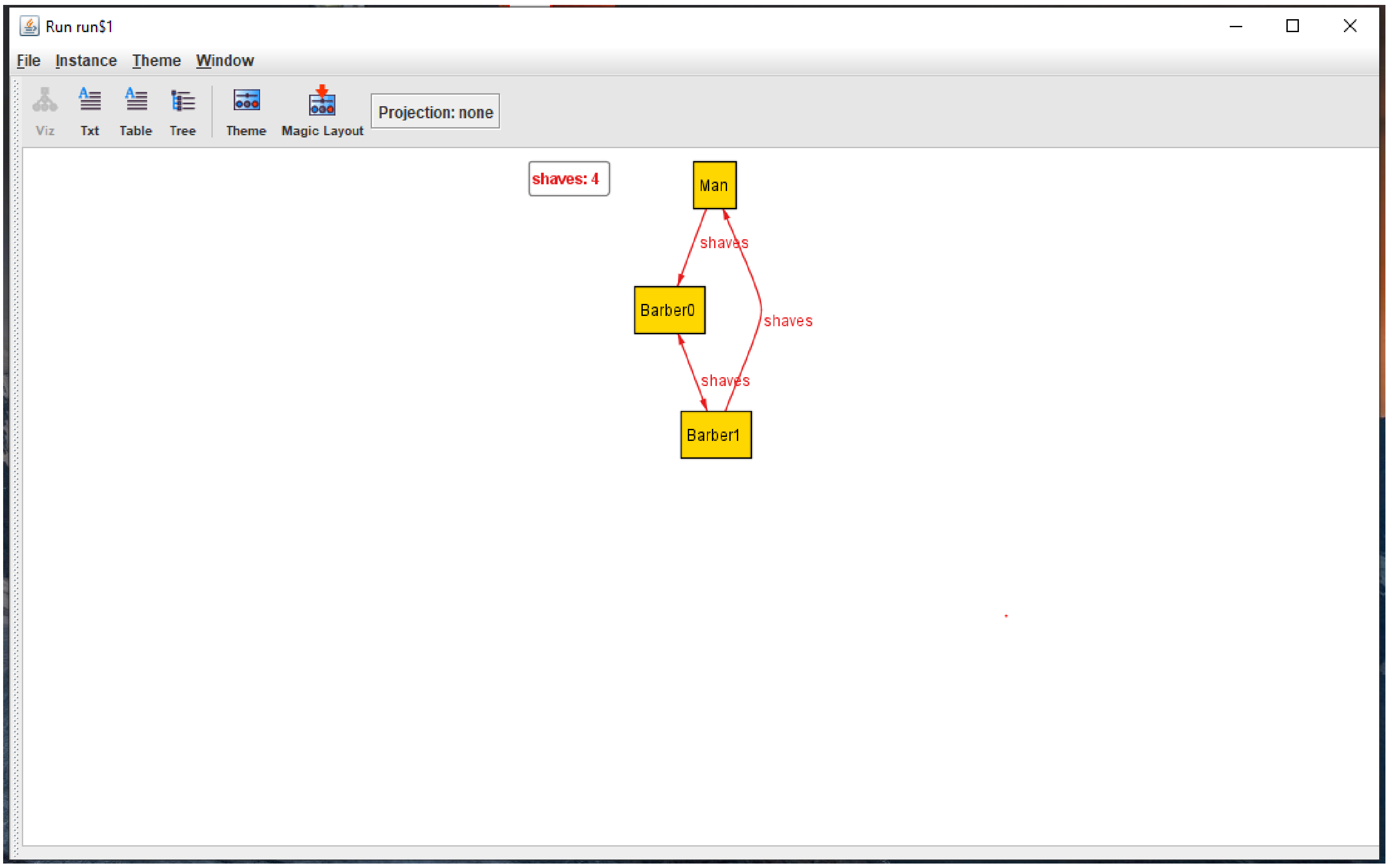


Figure 9 shows how the application invokes the Alloy API's GUI Tool to visualize instance information as a graphical graph. Users can also run the application with a specific command with an xml file to visualize the instance (see section 4.2 for the visualization command)

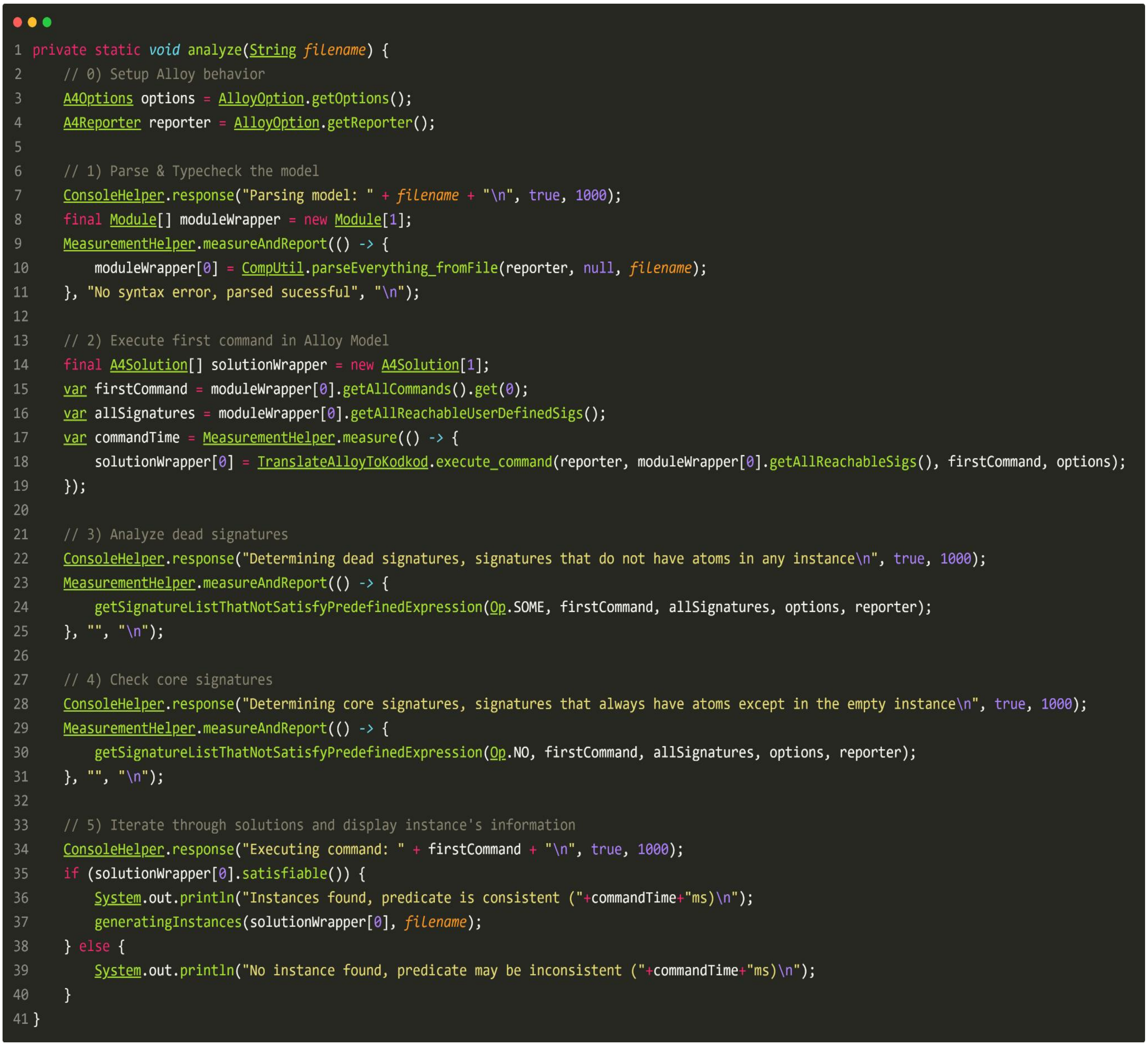
4 Implementation

We implemented our console application in Java and used the Alloy tool API as the main dependency for validating and analyzing the Alloy model input. Our implementation consists of five modules, each responsible for different tasks. All modules are located in the "app/src/main/java/alloy/interactive/teaching/material" directory and are organized by folder structure. Here's an overview of each module and its associated classes:

- Config module - This module stores Alloy's lesson config, which acts as a source of truth for the entire project. Lesson config allows us to introduce new lessons seamlessly by modifying only one place of code in the project. Once lesson resources (e.g., explanation files, exercise files, solution files) are correctly added to the right directory with the right structure, new lessons are seamlessly integrated into the application. (see Figure 12 and Figure 13)
- Helper module - This module contains helper classes that facilitate different functionalities. For example, the MeasurementHelper class measures performance time and reports the results to the console (see Figure 6). The FileHelper class reads lesson resources, displays them on the console, and saves all content to a new file for future reference. Figure 2 shows an example result from reading a lesson resource.
- Validator module - This module contains different validator classes used in the project. For instance, the CommandLineValidator class verifies the correctness of command arguments provided by the user.
- Alloy validator module - This sub-module communicates with the Alloy API, sets up its behavior for the project, validates the user's Alloy model, analyzes it, provides informative feedback to the console, and visualizes instance information in the form of text and graphical representation. Figure 7 and Figure 8 are examples of results produced by this module.
- App.java - This is the main class for the project, and it invokes several classes from different modules to produce proper user behavior in each flow (Section 3.1 and Section 3.2) of the application.

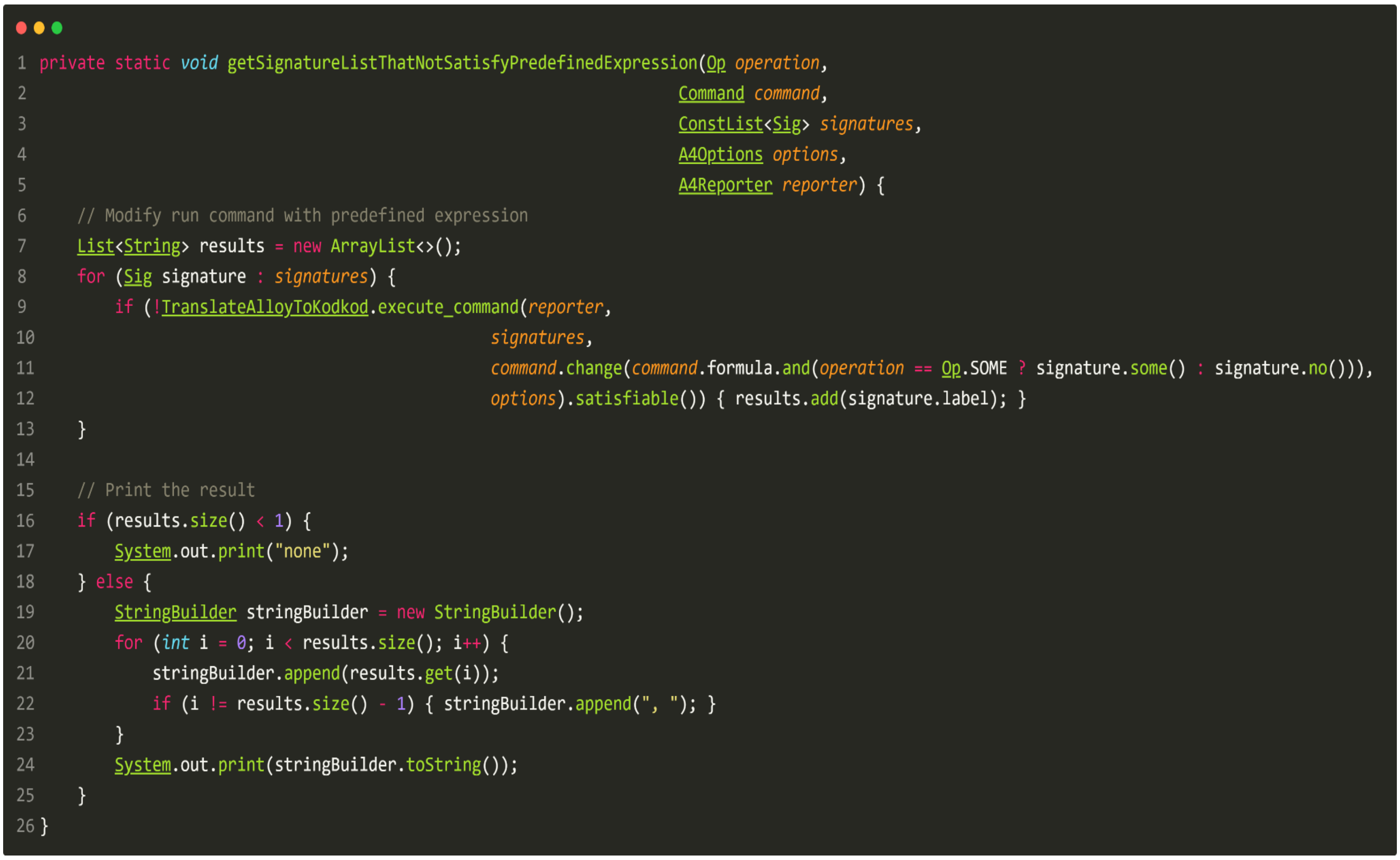
4.1 Code Example

We are proud to present some interesting pieces of code we developed in this project. However, for the sake of conciseness, we will only showcase code examples that demonstrate solutions to the challenges presented in Section 2.



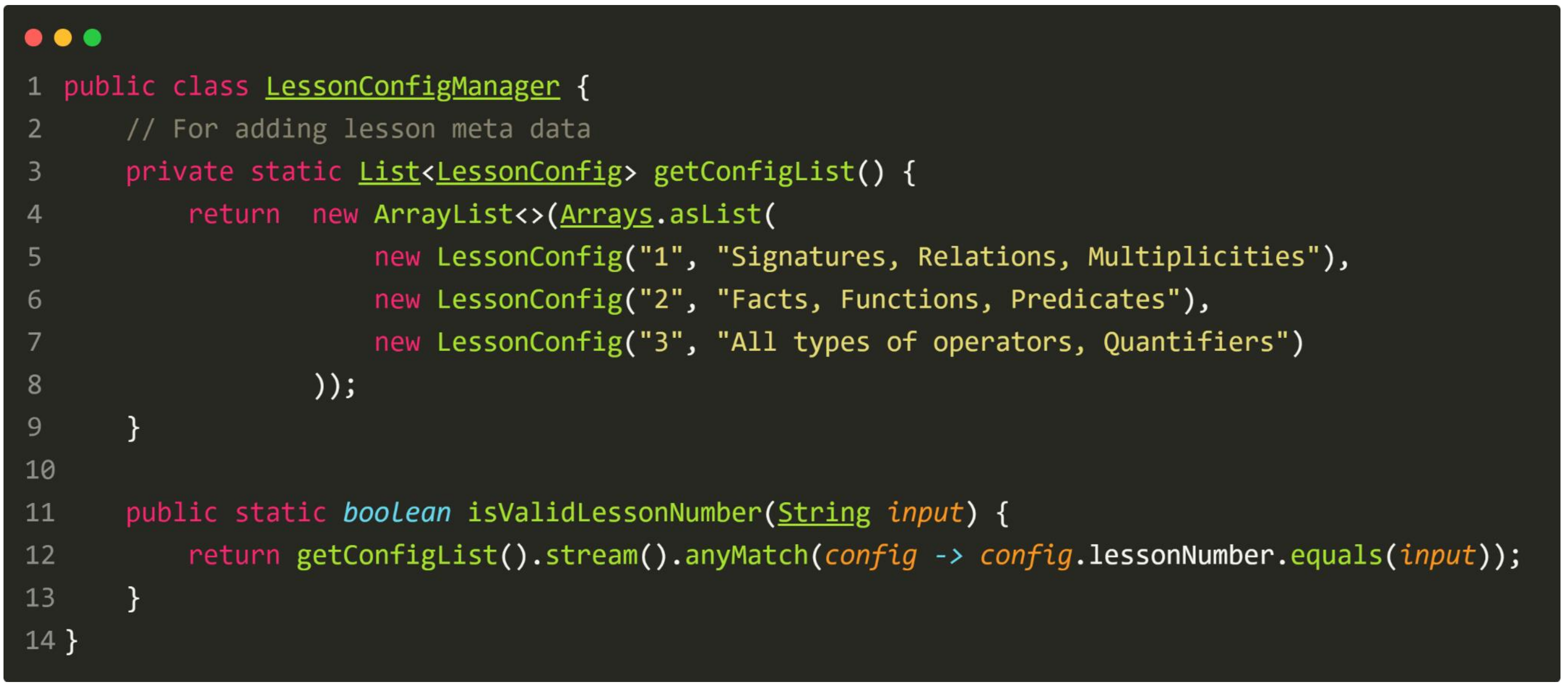
```java
private static void analyze(String filename) {
    // 0) Setup Alloy behavior
    A4Options options = AlloyOption.getOptions();
    A4Reporter reporter = AlloyOption.getReporter();

    // 1) Parse & Typecheck the model
    ConsoleHelper.response("Parsing model: " + filename + "\n", true, 1000);
    final Module[] moduleWrapper = new Module[1];
    MeasurementHelper.measureAndReport(() -> {
        moduleWrapper[0] = CompUtil.parseEverything_fromFile(reporter, null, filename);
    }, "No syntax error, parsed sucessful", "\n");

    // 2) Execute first command in Alloy Model
    final A4Solution[] solutionWrapper = new A4Solution[1];
    var firstCommand = moduleWrapper[0].getAllCommands().get(0);
    var allSignatures = moduleWrapper[0].getAllReachableUserDefinedSigs();
    var commandTime = MeasurementHelper.measure(() -> {
        solutionWrapper[0] = TranslateAlloyToKodkod.execute_command(reporter, moduleWrapper[0].getAllReachableSigs(), firstCommand, options);
    });

    // 3) Analyze dead signatures
    ConsoleHelper.response("Determining dead signatures, signatures that do not have atoms in any instance\n", true, 1000);
    MeasurementHelper.measureAndReport(() -> {
        getSignatureListThatNotSatisfyPredefinedExpression(Op.SOME, firstCommand, allSignatures, options, reporter);
    }, "", "\n");

    // 4) Check core signatures
    ConsoleHelper.response("Determining core signatures, signatures that always have atoms except in the empty instance\n", true, 1000);
    MeasurementHelper.measureAndReport(() -> {
        getSignatureListThatNotSatisfyPredefinedExpression(Op.NO, firstCommand, allSignatures, options, reporter);
    }, "", "\n");

    // 5) Iterate through solutions and display instance's information
    ConsoleHelper.response("Executing command: " + firstCommand + "\n", true, 1000);
    if (solutionWrapper[0].satisfiable()) {
        System.out.println("Instances found, predicate is consistent ("+commandTime+"ms)\n");
        generatingInstances(solutionWrapper[0], filename);
    } else {
        System.out.println("No instance found, predicate may be inconsistent ("+commandTime+"ms)\n");
    }
}
```

Figure 10 illustrates the analyze() method in the AlloyValidator class, which reflects the process of providing learners with informative feedback and solutions as they work on each lesson's exercise (challenge 2.2 in Section 2). When a user submits their Alloy solution as an .als file, the application invokes the Alloy API to perform several checks and analyses. These include type and syntax checks, analysis of dead and core signatures within the input model and display of instance information. Additionally, as part of challenge 2.3 in Section 2, we measure the performance time of each crucial process, as shown in lines 9, 23, and 29. This ensures that users do not have to wait too long and receive a response within a reasonable timeframe



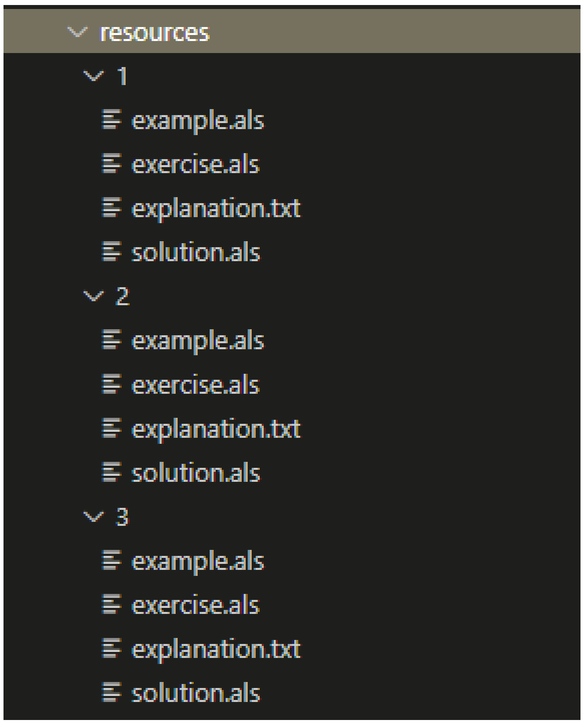
```java
private static void getSignatureListThatNotSatisfyPredefinedExpression(Op operation,
                                                                    Command command,
                                                                    ConstList<Sig> signatures,
                                                                    A4Options options,
                                                                    A4Reporter reporter) {
    // Modify run command with predefined expression
    List<String> results = new ArrayList<>();
    for (Sig signature : signatures) {
        if (!TranslateAlloyToKodkod.execute_command(reporter,
                                                signatures,
                                                command.change(command.formula.and(operation == Op.SOME ? signature.some() : signature.no())),
                                                options).satisfiable()) { results.add(signature.label); }
    }

    // Print the result
    if (results.size() < 1) {
        System.out.print("none");
    } else {
        StringBuilder stringBuilder = new StringBuilder();
        for (int i = 0; i < results.size(); i++) {
            stringBuilder.append(results.get(i));
            if (i != results.size() - 1) { stringBuilder.append(", "); }
        }
        System.out.print(stringBuilder.toString());
    }
}
```

Figure 11 provides a code snippet that analyzes the list of dead signatures and core signatures based on the run command in the user's Alloy model.



```java
public class LessonConfigManager {
    // For adding lesson meta data
    private static List<LessonConfig> getConfigList() {
        return  new ArrayList<>(Arrays.asList(
                    new LessonConfig("1", "Signatures, Relations, Multiplicities"),
                    new LessonConfig("2", "Facts, Functions, Predicates"),
                    new LessonConfig("3", "All types of operators, Quantifiers")
                ));
    }

    public static boolean isValidLessonNumber(String input) {
        return getConfigList().stream().anyMatch(config -> config.lessonNumber.equals(input));
    }
}
```

Figure 12 showcase the solution to challenge 2.4 in Section 2



```
resources
  1
    example.als
    exercise.als
    explanation.txt
    solution.als
  2
    example.als
    exercise.als
    explanation.txt
    solution.als
  3
    example.als
    exercise.als
    explanation.txt
    solution.als
```

Figure 13 showcase the solution to challenge 2.4 in Section 2

In Figure 12 and Figure 13, our lessons can be easily added or expanded by adding one line of code (lesson metadata) to the LessonConfigManager class, along with the lesson resources in the appropriate directory. New lessons are seamlessly populated into our application within minutes.

4.2 Tool Usage

Our tool requires Java 11 or later to run. To run our application, users can clone our repository at https://github.com/kingdomax/alloy-interactive-teaching-material and run the main class

"app\src\main\java\alloy\interactive\teaching\material\App.java"

Alternatively, we also provide a JAR file that is executable on any platform with Java 11 installed. Users have two options to run the application:

a) Walkthrough each unit lesson - This option allows users to navigate and learn through desired lessons. Upon completion of the lesson, users will be prompted with a syntax example and an exercise to test their knowledge. To run the application in this scenario, use the following command interface:

“java -jar alloy-interactive-teaching-material.jar”

b) Jump to a specific lesson's section - Users can quickly access a particular lesson's section by instructing the application with the following command synopsis:

"java -jar alloy-interactive-teaching-material.jar --lesson <lesson-number> --part <explanation||example||exercise||exercise-solution||exercise-submit> [--partParam <solution-file.als||instance.file.xml>]"

For example:

- "java -jar alloy-interactive-teaching-material.jar --lesson 2 --part explanation" - This command instructs the application to display only the explanation section from lesson 2.
- "java -jar alloy-interactive-teaching-material.jar --lesson 2 --part exercise" - This command instructs the application to display only the exercise section from lesson 2.
- ● "java -jar alloy-interactive-teaching-material.jar --lesson 2 --part exercise-solution" - This command instructs the application to display only the exercise's solution from lesson 2.
- "java -jar alloy-interactive-teaching-material.jar --lesson 2 --part exercise-submit my-solution.als" - This command instructs the Alloy analyzer to validate the "my-solution.als" model and display the feedback output.
- "java -jar alloy-interactive-teaching-material.jar --lesson 2 --part exercise-submit my-instance.xml" - This command instructs the application to visualize the instance from "my-instance.xml" in GUI mode.

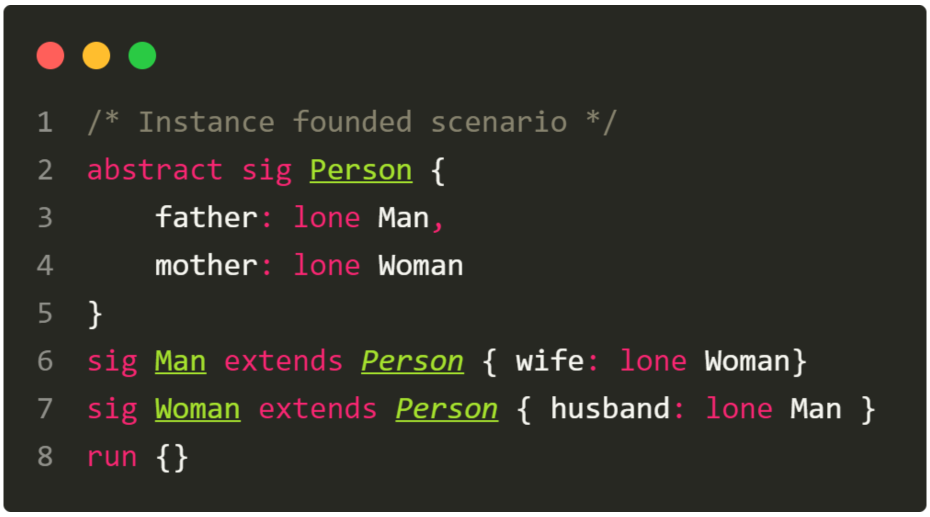
5 Evaluation

In this section, we will evaluate the performance and accuracy of our application in validating and analyzing the user's Alloy input model.

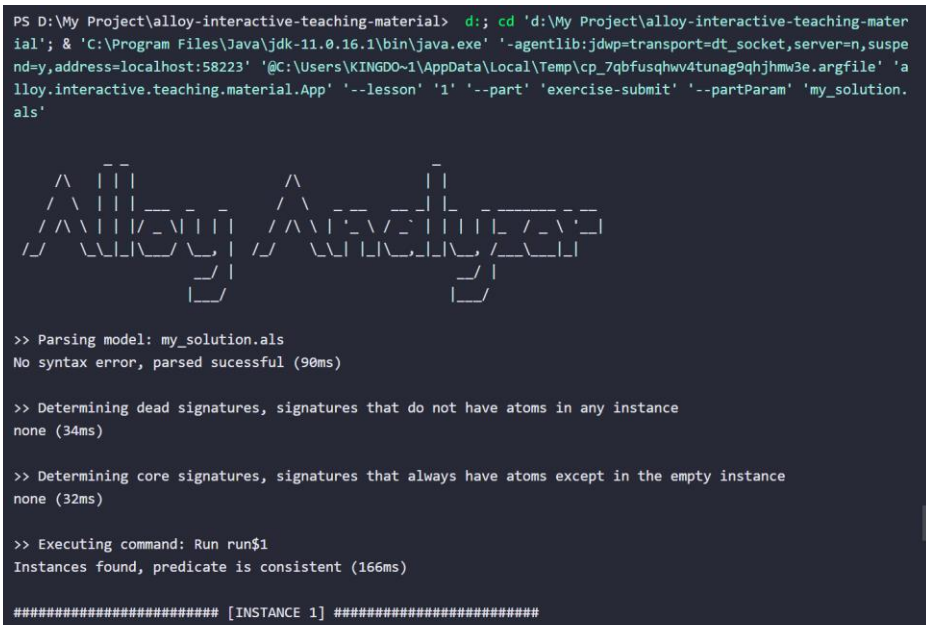
5.1 Validation

We have extensively tested our application on various Alloy model files and ensured that it produces the same results as using Alloy Tool directly. We have categorized the models into three scenarios based on their characteristics, and the results are shown below. (example of these scenarios can be found in the "my_solution.als" file)

a) Instance Found Scenario
In this scenario, the predicate is consistent, and the application successfully finds an instance that satisfies the constraints of the input model.



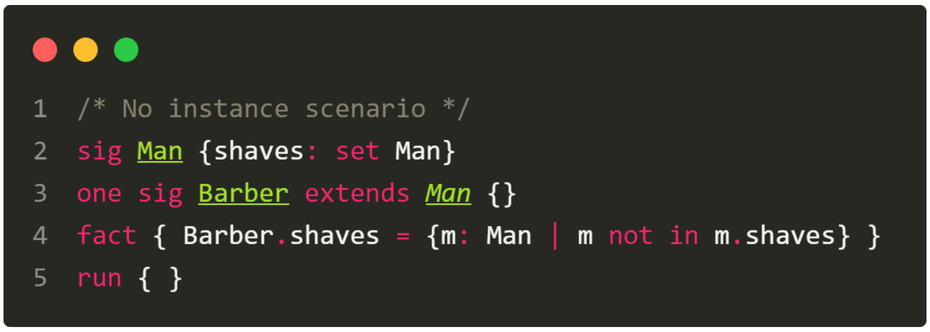
```
/* Instance founded scenario */
abstract sig Person {
    father: lone Man,
    mother: lone Woman
}
sig Man extends Person { wife: lone Woman}
sig Woman extends Person { husband: lone Man }
run {}
```



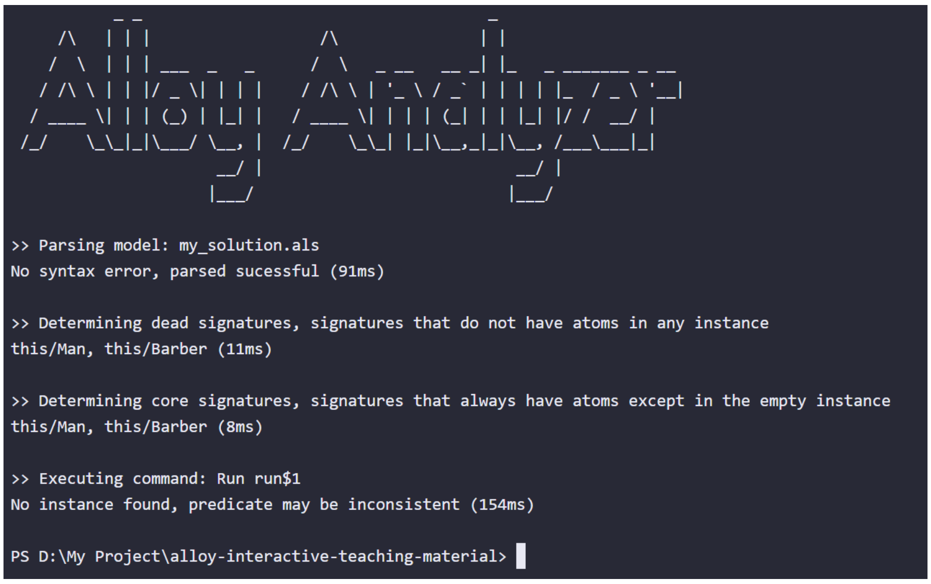
```
PS D:\My Project\alloy-interactive-teaching-material>  d:; cd 'd:\My Project\alloy-interactive-teaching-mater
ial'; & 'C:\Program Files\Java\jdk-11.0.16.1\bin\java.exe' '-agentlib:jdwp=transport=dt_socket,server=n,suspe
nd=y,address=localhost:58223' '@C:\Users\KINGDO~1\AppData\Local\Temp\cp_7qbfusqhwv4tunag9qhjhmw3e.argfile' 'a
lloy.interactive.teaching.material.App' '--lesson' '1' '--part' 'exercise-submit' '--partParam' 'my_solution.
als'


          _ _                                     _
    /\   | | |                /\                 | |
   /  \  | | | ___  _   _    /  \   _ __   __ _| |_   _ _______ _ __
  / /\ \ | | |/ _ \| | | |  / /\ \ | '_ \ / _` | | | | |_  / _ \ '__|
 /_/    \_\_|_|\___/ \__, | /_/    \_\_| |_|\__,_|_|\__, /___\___|_|
                     __/ |                         __/ |
                    |___/                         |___/

>> Parsing model: my_solution.als
No syntax error, parsed sucessful (90ms)

>> Determining dead signatures, signatures that do not have atoms in any instance
none (34ms)

>> Determining core signatures, signatures that always have atoms except in the empty instance
none (32ms)

>> Executing command: Run run$1
Instances found, predicate is consistent (166ms)

######################### [INSTANCE 1] #########################
```

b) Instance Not Found Scenario

In this scenario, the predicate is inconsistent, and the application cannot find any instance that satisfies the constraints of the input model. The application also detects any dead signatures within the input model

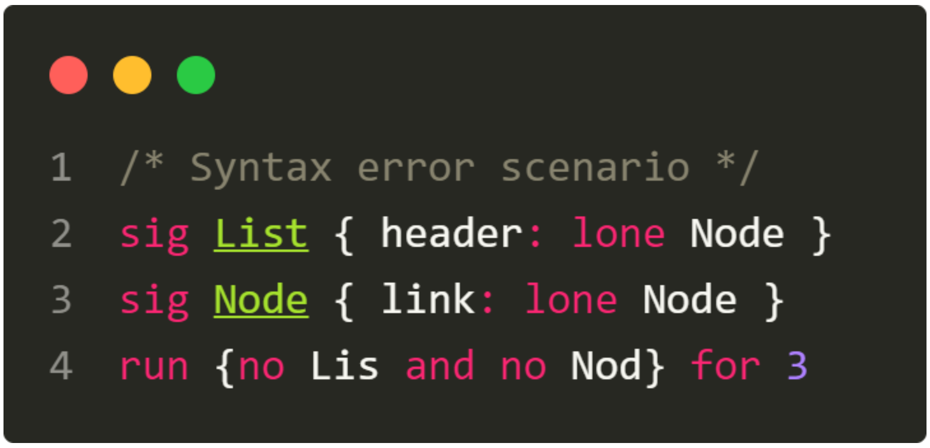


```
/* No instance scenario */
sig Man {shaves: set Man}
one sig Barber extends Man {}
fact { Barber.shaves = {m: Man | m not in m.shaves} }
run { }
```

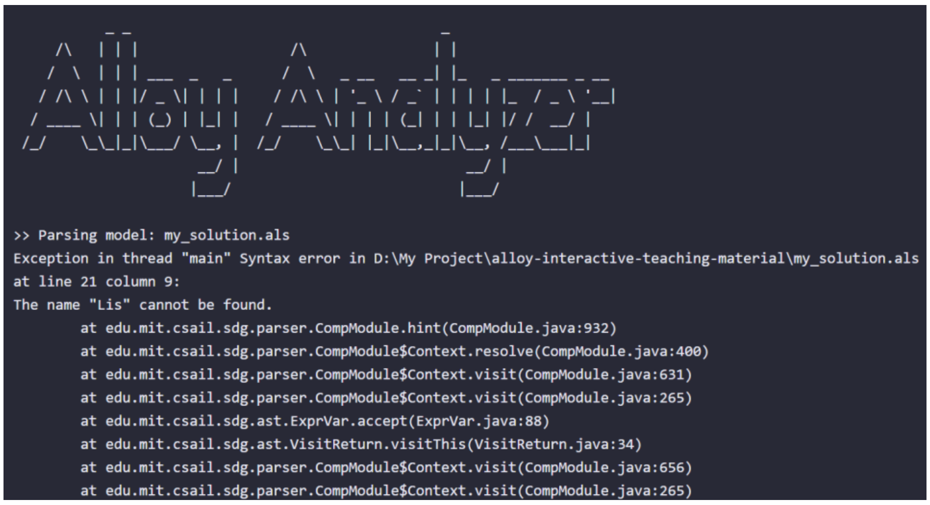


```
          _ _                                 _                    
    /\   | | |                /\             | |                   
   /  \  | | | ___  _   _    /  \   _ __   __ _| |_   _ _______ _ __ 
  / /\ \ | | |/ _ \| | | |  / /\ \ | '_ \ / _` | | | | |_  / _ \ '__|
 / ____ \| | | (_) | |_| | / ____ \| | | | (_| | | |_| |/ /  __/ |   
/_/    \_\_|_|\___/ \__, |/_/    \_\_| |_|\__,_|_|\__, /___\___|_|   
                     __/ |                         __/ |             
                    |___/                         |___/              

>> Parsing model: my_solution.als
No syntax error, parsed sucessful (91ms)

>> Determining dead signatures, signatures that do not have atoms in any instance
this/Man, this/Barber (11ms)

>> Determining core signatures, signatures that always have atoms except in the empty instance
this/Man, this/Barber (8ms)

>> Executing command: Run run$1
No instance found, predicate may be inconsistent (154ms)

PS D:\My Project\alloy-interactive-teaching-material>
```

c) Parsing Error Scenario
In this scenario, the input model contains syntax errors that prevent it from being parsed correctly. The application detects and reports any parsing errors in the input model.



```
/* Syntax error scenario */
sig List { header: lone Node }
sig Node { link: lone Node }
run {no Lis and no Nod} for 3
```



```
          _ _                                   _
    /\   | | |                 /\              | |
   /  \  | | | ___  _   _     /  \   _ __   __ _| |_   _ _______ _ __
  / /\ \ | | |/ _ \| | | |   / /\ \ | '_ \ / _` | | | | |_  / _ \ '__|
 / ____ \| | | (_) | |_| |  / ____ \| | | | (_| | | |_| |/ /  __/ |
/_/    \_\_|_|\___/ \__, | /_/    \_\_| |_|\__,_|_|\__, /___\___|_|
                     __/ |                          __/ |
                    |___/                          |___/

>> Parsing model: my_solution.als
Exception in thread "main" Syntax error in D:\My Project\alloy-interactive-teaching-material\my_solution.als
at line 21 column 9:
The name "Lis" cannot be found.
        at edu.mit.csail.sdg.parser.CompModule.hint(CompModule.java:932)
        at edu.mit.csail.sdg.parser.CompModule$Context.resolve(CompModule.java:400)
        at edu.mit.csail.sdg.parser.CompModule$Context.visit(CompModule.java:631)
        at edu.mit.csail.sdg.parser.CompModule$Context.visit(CompModule.java:265)
        at edu.mit.csail.sdg.ast.ExprVar.accept(ExprVar.java:88)
        at edu.mit.csail.sdg.ast.VisitReturn.visitThis(VisitReturn.java:34)
        at edu.mit.csail.sdg.parser.CompModule$Context.visit(CompModule.java:656)
        at edu.mit.csail.sdg.parser.CompModule$Context.visit(CompModule.java:265)
```

5.2 Performance Cost

We have measured the performance cost of parsing the Alloy model, analyzing the Alloy model, and executing the Alloy command. Our test models were obtained from the example files and exercise files in the project's "app\src\main\resources" directory. We measured the execution times of the validation checks described above. All measurements were performed on a laptop computer with an Intel i7 processor running at 3.6GHz (11th generation). We have computed the average execution times among all test models as ~400-500ms. Given the reasonable execution times, we did not need to further optimize the application.

6 Conclusion

In this report, we have presented the development of an interactive teaching application for the Alloy language, addressing the challenges of the language's steep learning curve and lack of resources for beginners. Our application provides a unit lesson with examples, practice exercises, and informative feedback, enabling learners to gain a solid understanding of the language's key concepts at their own pace.

We have identified several challenges in developing this application, including the creation of efficient lessons and exercises, informative feedback provision, performance consideration, and modular design. To address these challenges, we have developed a lesson flow that allows users to navigate and learn through desired lessons and an Alloy analyzer flow that checks and analyzes user input using the Alloy API library.

Our initial implementation covers three unit lessons, each focusing on specific topics, and enables users to revisit the lesson sections later. The application provides performance measurements and visualizes instance information as a graphical graph, further assisting learners in understanding the language's concepts.

In conclusion, our interactive teaching application for the Alloy language provides a user-friendly and effective approach to learning Alloy, reducing the barrier to entry and increasing accessibility for beginners. Future work includes adding new functionality and features without disrupting the existing codebase, further improving the application's performance, and expanding the lessons' scope to cover more advanced topics.