GreenMirror: A Visualization and Animation Framework for State-Transition Models

Bachelor assignment

K.M. El Assal (s1097539)
University of Twente
P.O. Box 217, 7500AE Enschede, the Netherlands
`k.m.elassal@student.utwente.nl`

Exam committee:
prof.dr.ir. A. Rensink
dr. H.K. Hemmes

repository: `https://github.com/iisys-/GreenMirror`

August 13, 2015

Abstract

State-transition models are often used to analyse the discrete dynamic behaviour of systems, although analysis may prove to be tedious when a system has a huge amount of possible states. A tool is being developed using the Java programming language and the JavaFX library to provide a different approach. Continuing previous work, this report describes the next step that has been made in the development of a framework that can visualize a system's states and animate transitions that result from a list of state-transitions. The user can define complex state-transition models during runtime which will be visualized according to the state-transitions the user also provides during runtime. The framework has been specifically designed to support future development by being modular, extensible and maintainable.

Contents

1 Introduction

This report describes the GreenMirror framework: a tool for visualizing and animating state-transition models. The current section contains background information needed to understand this project and the rest of the report. Section 2 gives an example of the usage of the GreenMirror application. The example model is the ferryman puzzle and shows how one can switch from a state space generation tool to the GreenMirror tool. The features of the application and their usage are described in section 3, which is particularly useful for the end-users of GreenMirror. Section 4 follows with a description about how the design and implementation of the GreenMirror framework look like and is useful for future developers and also in part for tool owners. Validation of the system is described in section 5 and this report concludes with section 6 discussing the developed system and providing suggestions for future improvements. This report has been written for anyone who wants to use the GreenMirror tool, which is why the current structure has been chosen: first can be seen why the system has been developed, then what the possibilities of the system are and finally how the system is developed and how the project was defined.

1.1 Background

State-transition models describe the discrete dynamic behaviour of a system or process. Such a model defines how the state of a system changes as the result of a trigger, and following its set of model rules. In the field of computer science, these models are used for system verification and analysis. Verification and analysis are done, for example, to confirm and prove a process can not enter a deadlock state. One way of analysing a state-transition model is by analysing its *state space*, which a state space generation tool such as GROOVE [10] can generate.

State spaces can become huge, complex and difficult to analyse. The research group Formal Methods and Tools[1] of the University of Twente has set out to develop a framework based on the Java programming language and the JavaFX library that enables researchers to analyse state-transition models using a different approach. The idea is to visualize the system state and animate the state-transitions of a user-specified state-transition model with the goal of gaining a better understanding of the model and its flaws.

The first step in the development of this framework has been made by Alex Aalbertsberg [1]. This report describes how his framework has been redesigned and how it has implemented more extensive features. The current version has been named *GreenMirror*. Although the project is far from complete, hopefully these contributions will eventually lead to better state-transition model research.

1.2 Glossary

Some terms in this report can be ambiguous or unclear in their meaning. For that reason, the following glossary is provided which will make their meaning definitive.

The application This refers to the GreenMirror application as used by the user to reach the goal as intended by the developer. It is a compiled version of the source code that is the GreenMirror framework.

The framework This refers to the application in development and specifically its source code.

GreenMirror The name of this project. It does not have a profound meaning, it is chosen to be a short designation to indicate that this project is meant in certain contexts.

[1] `http://fmt.cs.utwente.nl/`

FX This refers to the visual appearance linked to a GreenMirror node (which can, but does not per se equal the corresponding JavaFX node).

JavaFX The visualization library of the Java programming language.

JavaFX node A visual node of JavaFX. See "node".

JavaFX transition The animation of a JavaFX node property from one value to another. A JavaFX transition is always meant when talking about an animation in a JavaFX context.

The (researched/user-defined) model The model that the user wants to visualize and research using the GreenMirror application.

(GreenMirror) node The term "node" is used for two types of nodes: for a node in the model: a GreenMirror node; and a node of JavaFX. When speaking simply of a node, a GreenMirror node is intended. In any other case, the words "JavaFX node" will be used. It is also possible both a GreenMirror and its corresponding JavaFX node are meant. In that case, it will be clear from the context. It shall be made clear explicitly if other types of nodes are meant anywhere in this report.

State The state of a user-defined model.

State-transition A transition from one state of the user-defined model to another. In the visualizer, this can contain multiple JavaFX transitions.

State-transition model A model that describes how a system transitions from one state to another.

Trace A sequence of state-transitions.

User The user is the researcher that uses the framework to visualize certain models (see section 1.3).

Visualizer The component of the GreenMirror application that actually visualizes the states and the state-transitions.

1.3 Stakeholders

It is assumed that any stakeholder knows what he is doing when interacting with the system in the sense that the stakeholder knows about the context and the tools he is using. There are three ways of interaction, so the groups of stakeholders are divided as such.

The user The end-user, or "user" for short, is the stakeholder that uses the application for its main purpose: visualizing models. The "user" can also be renamed "the visualization builder", but in this report the term "user" will be retained. The user is expected to have a basic understanding of programming in general and specifically of the Java language. The user can also be one who only sees the visualization, but does not build it, in which case he can be renamed "the visualization viewer". This type of user will be considered the same as the "end-user" and will be taken into account by making the user interface of the visualization as straightforward as possible.

The state space tool owner Researchers might want to connect their own state space tool to the GreenMirror framework. The development of a suitable interface for this is taken into account in the design of the framework. Any person that writes a new extension to connect his state space tool to the GreenMirror framework is called a "state space tool owner", or "tool owner" for short, and will be considered in the design and development phases.

The developer The GreenMirror framework is intended to be extended and otherwise improved over time. The developers that will do so are also seen as a relatively small group of stakeholders. To extend and improve the framework developers will need an in-depth understanding of the framework, contrary to tool owners who only need to understand the interface between their tool and the framework. Hence, large part of this report is written for developers.

1.4 Project definition

The main goal of this project is to make it possible for researchers to visually analyse the temporal behaviour of state-transition models. These models are already defined and might have already been visualized and analysed in a different way. Several requirements and use cases have been defined that mark the scope of this project. Furthermore, three test cases have been chosen that GreenMirror must be able to visualize. The ultimate goal for a final version of this tool is to be able to create and alter state-transition models from a visualizer or other kind of graphical user interface. This means that not only should the tool turn a model into visualizations, but also it should also, eventually, be able to turn visualizations into a model. This unfortunately lies beyond the scope of this project.

The requirements of this project have been divided into architectural, functional and performance requirements and are listed in table 1. The architectural requirements are defined for the framework, so they are also inherent to the application. The developer is the main stakeholder concerning the architectural requirements, since he has most to gain from a well-defined software system. The tool owner is also an important stakeholder, but mostly in the extensibility requirement. The architectural requirements are fairly general and vague, but are important and have to be stated nevertheless. All functional requirements relate to the application, seeing as the application contains functionality and the framework does not. Therefore, the user is the main stakeholder in these requirements. In general, the application has to be *flexible*. The user is assumed to be a researcher, which implies that the user wants to have as much freedom as possible in deciding what the application does, how it works and what it gives as output. This notion is the basis for all functional requirements. The performance requirements also relate to the application and primarily concern the smooth and uninterrupted execution of visualizations. All requirements are formulated, sorted (in groups of the requirement type) and prioritized according to the *MoSCoW* method.

Table 1: project GreenMirror requirements

Architectural requirements	
Req. 1	***The framework must be easily extensible.*** Future research might require new functionalities, so a developer or tool owner must be able to extend easily instead of heavily modify the source code. Fayad & Schmidt [3] call these extensibility points "hot spots". Use case 1: as a developer (or tool owner), I want to be able to extend the framework with as less source code alterations as possible.
Req. 2	***The framework must be maintainable.*** A developer must have little to no effort in understanding the source code when improvements or extensions are developed. Programmed structures and patterns must be easily recognizable and well documented.

Functional requirements	
Req. 3	***The application must become aware of the researched model*** and how to visualize it. This process is divided into several parts, although it still treated as one requirement. – The application must become aware of the initial nodes and relations the researched model is composed of. – The application must become aware of how these initial nodes and relations should be visualized. – The application must become aware of how state-transitions should be visualized. Use case 2: as a user, I want to choose how the application becomes aware of my model. Use case 3: as a user, I want to choose the source the application retrieves my model from.
Req. 4	***The application must become aware of the trace***. Use case 4: as a user, I want to choose the source that the application retrieves my trace from.
Req. 5	***The application must generate visualizations of the user's model progressing through the state-transitions on the trace.*** Additionally, the following sub-requirements are defined: – Req. 6: ***the application must visualize simple geometric shapes***, such as rectangles and circles. – Req. 7: ***the application must visualize text nodes***. – Req. 8: ***the application must visualize images***. – Req. 9: ***the application must visualize simple animations***, such as node movement and creation. – Req. 10: ***the application must visualize the placement of nodes with respect to other nodes***, without receiving coordinates from the user. Use case 5: as a user, I want to view the visualizations that the application generated from my model.
Req. 11	***The application should provide a detailed log*** about all relevant events. This helps in debugging, in improving the application and in analysing the researched model. Use case 6: as a user, I want to view a detailed log.
Req. 12	***The application should be able to browse back and forth between the visualized states***, while consistently seeing the proper visualization and without errors or reduced performance. This way the user doesn't have to rerun the complete application on each examination of the model. Use case 7: as a user, I want to browse back and forth between the visualized states of my model.

Performance requirements	
Req. 13	***The application should transition from state to state without noticeable delay*** caused by memory or processing problems. This requirement only applies to the point in time where the model has been completely loaded into GreenMirror.
Req. 14	***The application should not crash*** or terminate otherwise while transitioning from state to state. This requirement only applies to the point in time where the model has been successfully loaded into GreenMirror and all model logic has been deemed valid and without errors.

The first test case is the ferryman puzzle as discussed in section 2. The second test case is the well-known game ConnectFour. The third is the *Dining Philosophers problem* (Dijkstra [2]), which is often used to illustrate concurrency problems such as shared resources and deadlock scenarios. The problem scenario consists of a certain amount of forks and an equal amount of philosophers that sit around a table. Each philosopher has a plate of spaghetti in front of him and each pair of philosophers has a fork between them. The goal is to come up with a fair way for each philosopher to eat and not die of starvation. The following constraints are in place.

1. Philosophers do only one of three things: think, be hungry or eat. They do not communicate with each other.
2. A philosopher needs two forks to eat and can only use the two forks on his immediate sides.
3. When a philosopher is thinking, he does not have forks and does not do anything at all.
4. When a philosopher gets hungry, he tries to obtain the two forks on his sides and will wait for their availability. He will not put down his first fork before he gets to eat.
5. After a philosopher is done eating, he releases the two forks.

The forks represent shared resources in a concurrency problem. The constraints represent synchronization measures. A deadlock can, for example, occur when all philosophers get hungry at the same time and pick up the fork on their right.

2 The ferryman

The following is an example of a simple state-transition model sometimes used to illustrate the concept of state-transition models [6]. First, the model configuration is shown in the state space generation tool GROOVE. Then the same model is shown in a configuration that can be interpreted by GreenMirror. Finally, the result from the GreenMirror visualization is shown. This demonstrates the basic features, how GreenMirror can be used and how simple its usage can be.

The ferryman puzzle consists of four active objects: the wolf, the goat, the cabbage and the ferryman. In the initial state of the system, the first three objects are on the left bank of a river. The ferryman is moored to the same bank with his ferry. The goal is to transport the wolf, the goat and the cabbage to the right bank of the river without the wolf eating the goat or the goat eating the cabbage. The following rules apply.

1. The ferryman can only bring one passenger at a time.
2. If the wolf and the goat are together on either side of the river without the ferryman present, the wolf eats the goat.
3. If the goat and the cabbage are together on either side of the river without the ferryman present, the goat eats the cabbage.

GROOVE uses *grammars* to describe the rules of a model. The grammar describing the initial state of the ferryman model is seen in figure 1. `Wolf`, `Goat` and `Cabbage` are subtypes of the `Cargo` type. The grammars for the final states are omitted here because they are less relevant than those shown in figures 1 and 2.

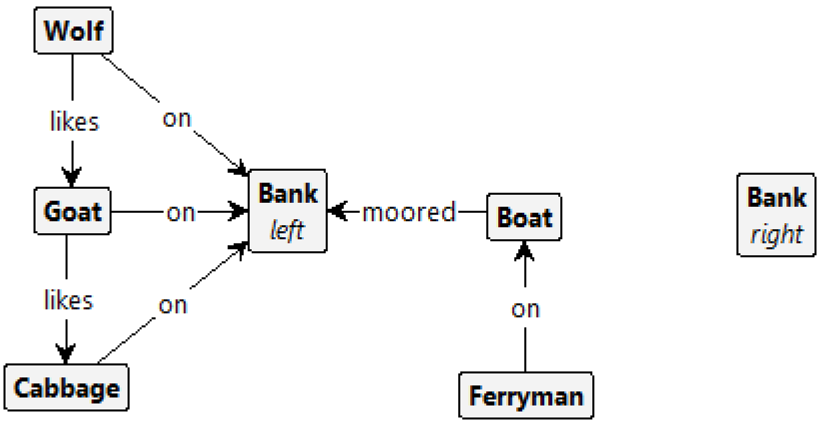
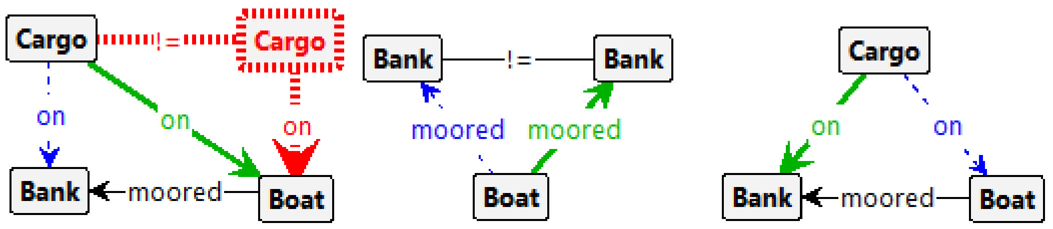


Figure 1: GROOVE grammar of the initial state of the ferryman model

The state space generated from this model is seen in figure 3. This state space contains 35 states and 70 transitions and is excellent for analysing state-transitions needed to get to a specific state. However, if one wants to visualize and animate the state-transitions that result in a specific state, these space generations tools don't have much to offer. GreenMirror can visualize this relatively easy.

This model can be translated easily into files that can be interpreted by GreenMirror. For this example, the Groovy script model initializer and file trace selector implementations will be used (see section 4.4). Listing 1 shows how the model is initialized. On line 1, the visualizer is initialized with a width of 500 pixels, a height of 300 pixels and a default JavaFX transition



(a) transition: load (b) transition: cross (c) transition: unload

Figure 2: GROOVE grammars for state-transitions of the ferryman model. The `Cargo` and `Bank` nodes are types: they indicate any cargo or river bank, respectively, that is selected with the state-transition. After the state-transition is complete, edges with a dashed blue line are removed and those with a solid green line are created. The red node and edge means: "if `Boat` has no other `Cargo` on it."

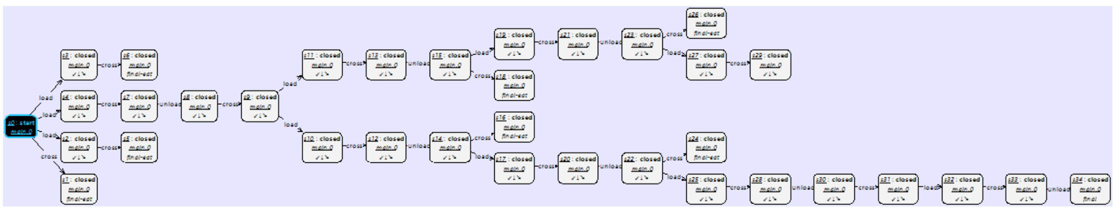


Figure 3: GROOVE generated state space of the ferryman model, using the breadth-first exploration strategy and final states acceptor, visualized using the compact tree layout and spanning tree filter. The blocks are the unique states and the labelled lines between them are the state-transitions. Note: because of the size of the image, the states and state-transitions are practically unreadable. This does not make the image any less relevant: it's an illustration of how a state space can become unclear easily and how it can be used to for analysis.

duration of 1000 milliseconds. On lines 3 to 46, the initial state is defined: the background, the ferry, the wolf, the goat and the cabbage nodes and all relations between them are created. These lines are roughly equivalent to figure 1, with additional visualization information. Lines 49 to 58 are equivalent to figure 2a. A clear difference lies in the conditional that the ferry can not already hold a cargo node: GROOVE simply does not explore the states resulting from that state-transition, whereas GreenMirror gives an exception and aborts when this state-transition is erroneously encountered on a trace. Lines 61 to 68 and 71 to 80 are equivalent to respectively figures 2b and 2c.

Listing 1: example Groovy code for the ferryman model

```
1  initialize(500, 300, 1000);
2
3  addNodes(
4      // Background.
5      new Node("bank:left")
6              .set(fx("rectangle").setSize(150, 300).setPosition(0, 0)
7                                          .setFill("linear-gradient(to␣right,␣darkgreen␣0%,␣
                                               limegreen␣92.5%,␣#00F0F0␣100%)")),
8      new Node("river")
9              .set(fx("rectangle").setSize(200, 300).setPosition(150, 0)
10                                         .setFill("linear-gradient(to␣right,␣#00F0F0␣0%,␣#00
                                               DEDE␣50%,␣#00F0F0␣100%)")),
11     new Node("bank:right")
12             .set(fx("rectangle").setSize(150, 300).setPosition(350, 0)
13                                         .setFill("linear-gradient(to␣left,␣darkgreen␣0%,␣
                                               limegreen␣92.5%,␣#00F0F0␣100%)")),
14     // Ferry.
15     new Node("ferry")
```

```
        .set(fx("image").setImageFromFile("testcases/img/boat.png")
                        .setFitWidth(100).setPreserveRatio(true)),
  // Cargo.
  new Node("cargo:goat")
        .set(fx("image").setImageFromFile("testcases/img/goat.png")
                       .setFitWidth(50).setPreserveRatio(true)),
  new Node("cargo:wolf")
        .set(fx("image").setImageFromFile("testcases/img/wolf.png")
                        .setFitWidth(50).setPreserveRatio(true)),
  new Node("cargo:cabb")
        .set(fx("image").setImageFromFile("testcases/img/cabbage.png")
                        .setFitWidth(50).setPreserveRatio(true)),
);

// Relations.
Relation onRelation = new Relation("on").setNodeB(node("bank:left"))
                                        .setPlacement(Placement.RANDOM);
addRelations(
  new Relation().setNodeA(node("ferry"))
                .setName("moored_to")
                .setNodeB(node("bank:left")).setPlacement(Placement.EDGE_RIGHT),
  onRelation.clone().setNodeA(node("cargo:goat")),
  onRelation.clone().setNodeA(node("cargo:wolf")),
  onRelation.clone().setNodeA(node("cargo:cabb")),
  new Relation().setNodeA(node("cargo:wolf"))
                .setName("likes")
                .setNodeB(node("cargo:goat")),
  new Relation().setNodeA(node("cargo:goat"))
                .setName("likes")
                .setNodeB(node("cargo:cabb"))
);

// Transition: load.
addTransition("load_(goat|wolf|cabb)", { String cargo ->
  if (node("ferry").getRelatedNodes(-1, "on").ofType("cargo").size() > 0) {
      fail("The␣ferry␣can␣only␣hold␣one␣cargo␣object.");
  }
  switchPlacementRelation(
    new Relation("on").setNodeA(node("cargo:" + cargo))
                      .setNodeB(node("ferry"))
                      .setPlacement(Placement.RANDOM).setRigid(true)
  );
});

// Transition: cross.
addTransition("cross", {
  switchPlacementRelation(
    node("ferry").getRelation(1, "moored_to")
      .clone()
      .setNextNodeB(nodes().ofType("bank"))
      .setNextPlacement(Placement.EDGE_RIGHT, Placement.EDGE_LEFT)
  );
});

// Transition: unload.
addTransition("unload", {
  for (Node cargo : node("ferry").getRelatedNodes(-1, "on").ofType("cargo")) {
    switchPlacementRelation(
      new Relation().setNodeA(cargo)
                    .setName("on")
                    .setNodeB(node("ferry").getRelatedNode(1, "moored_to"))
                    .setPlacement(Placement.RANDOM)
    );
  }
});
```

The trace chosen for this example is the shortest trace that leads to the successful solution of the ferryman puzzle. In figure 3, it is the path starting from the initial state, shown left in the figure, and ending in the state, shown in the bottom right corner of the figure. All other final states are unsuccessful solutions where either the goat or the cabbage gets eaten, or states that transition back into one of the displayed unique states. The trace is shown in listing 2.

Listing 2: example trace for the ferryman model

```
load_goat
cross
unload
cross
load_wolf
cross
unload
load_goat
cross
unload
load_cabb
cross
unload
cross
load_goat
cross
unload
```

The ferryman model and trace from respectively listings 1 and 2 result in the GreenMirror visualization of which the final state is shown in figure 4. With just 80 lines of code and a pre-defined trace, the ferryman model of figures 1 to 3 can be visualized and animated in GreenMirror for further analysis. It took GreenMirror about one second to interpret the model and the trace, followed by roughly five seconds to add all data to the visualizer after which the user could start moving through the model states. As is seen from the code in listing 1, images, geometric shapes such as rectangles and directed relations can be used. Furthermore, nodes can be placed according to their relations and complex (programming) logic can be added. Section 3 contains more information about the currently available features.

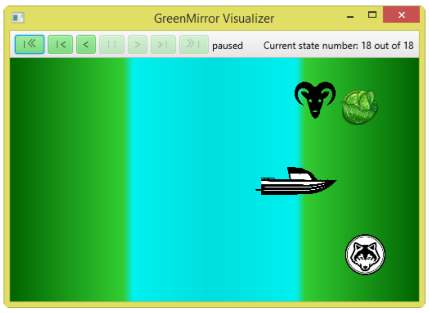


Figure 4: a screenshot of the by GreenMirror visualized final state of the ferryman model as defined in listings 1 and 2

3 Features

Knowledge of the features of GreenMirror is imperative when using the application. The two parts GreenMirror is composed of and their usage is first described in section 3.1. The way a model is interpreted is then elaborated upon in section 3.2, followed by a description of an integral part of the visualizations in section 3.3: node placement. Further possibilities are presented by discussing all currently available commands in section 3.4. Sections 3.5 and 3.6 briefly expand upon two auxiliary functionalities: the log and the GridBuilder class, respectively.

3.1 Client and server

The GreenMirror application is divided into two distinct components: the client and the server. The client interprets the user's model and translates it into commands. The server performs the actual visualization on the basis of these commands. Due to this decoupling, the framework is more maintainable and extensible and can thus be easily linked to other components or component versions.

Both components have to be executed using command line options. For the client to run, it needs a server to be available. To start the server, only one option has to be provided: the port. Use, for example, the option −−port=81 to run the server on port 81. The −−verbose option can be added to enable verbose output to the log and −−help to show all available command line options.

The client has three required options: −−host, −−model and −−trace. If a GreenMirror server is running local on port 81, the first option would look like −−host=127.0.0.1:81. To initialize a model using a Groovy script (which is currently the only supported model initializer), use −−model=groovyscript:<groovyfile> and to select a trace using the file selector (which is currently the only supported trace selector), use −−trace=file:<tracefile>. <∗file> should, of course, be replaced with the corresponding file names. Similar to the server, −−verbose and −−help can also be used with the client. For more details, see sections 4.2 to 4.4.

3.2 Nodes and relations

Tools such as GROOVE [10] use *nodes* and *edges* to define their model. This terminology can not simply be copied, because the definition and usage of an edge is slightly different than the equivalent entity used in GreenMirror. GreenMirror uses *nodes* and directional *relations* to define the model. State-transitions consist of adding and removing nodes, altering the appearance of nodes and changing relations between nodes.
Node properties include a type, a name, labels and an appearance wrapper, all of which are optional. When the node is added to the model, it receives an internal identification number; "ID" for short. The user does not have to interact with this ID in any way. Nodes also store their relations with other nodes.
Relations are always directional, going from "node A" to "node B". All relations have a name, a placement, a rigidity and a temporary appearance for node A, of which the latter lasts for the duration of the relation. These properties are optional, although it is recommended to always specify a name. There are two kinds of relations: placement relations, indicating that node A has a placement relative to node B on the visualizer, and non-placement relations, where the placement is set to NONE. The rigidity property can only be set for placement relations. When set to true it indicates that node A should follow when node B moves on the visualizer. If the rigidity is set to false, the placement is only calculated and applied when the relation is created: it won't be maintained when node B moves.

Each GreenMirror node that has a visual appearance needs to store and track the properties of its FX. This is internally done using implementations of the abstract FxWrapper class. The FX type can only be defined once for every GreenMirror node. The currently supported FX types are:

- rectangle;
- circle;
- text; and
- image (with a local or remote source).

Per FX type, certain properties can be set only initially and some can also be set or changed after the node's FX initialization. Due to the nature of GreenMirror, all properties of the latter type must be animatable. The `opacity` property, for example, is animatable by default by the JavaFX library. The `width` of a rectangle, on the other hand, is not animatable by default. GreenMirror includes animation support for some properties such as `width` so these properties can be changed during state-transitions. GreenMirror also supports the animation of some discrete properties, such as the `text` property of the text FX type. This is done by using a fast fade-out on the JavaFX node, changing the property and then using a fast fade-in. Support for additional FX types and FX type properties can be easily added to the framework. See section 4.6 and appendices A.3 and A.4.

3.3 Node placement

Placements are an important aspect of the visualization of nodes and their relations. They provide a level of abstraction and let the user worry rather about the model than about the actual coordinates of nodes on the visualizer. Placement relations between nodes indicate that one node of the relation, node A, is placed in a specific respect to the other node of the relation, node B. There are currently several extensions of the abstract Placement class implemented which are described below and are illustrated in figure 5. Every instance of Placement also has an optional position relative to the placement. For example: if a node A has an edge top placement with relative position $(0, -20)$ on node B, node A is placed 20 pixels above (and centred on) the edge of node B.

`Corner*Placement` A placement on any of the corners of a JavaFX node: top left, top right, bottom right or bottom left.

`Edge*Placement` A centred placement on any of the edges of a JavaFX node: top, right, bottom or left.

`MiddlePlacement` A placement in the exact middle of a JavaFX node.

`EdgePlacement` A placement on the edge of a JavaFX node, according to a specified angle. An angle of zero degrees is the equivalent an edge top placement, and an angle of 90 degrees (positive) is the equivalent of an edge right placement.

`CustomPlacement` A placement where only the relative position data is used to determine the coordinates. The relative position data is relative to the coordinates calculated for the MiddlePlacement.

`RandomPlacement` A random placement on a JavaFX node. Upon receiving this placement data, the server replaces this with a CustomPlacement where the relative position is set to the calculated, relative coordinates of the RandomPlacement. This is done so the random coordinates aren't recalculated every time node B moves (in the case of a rigid placement relation).

`NoPlacement` The default for a relation.

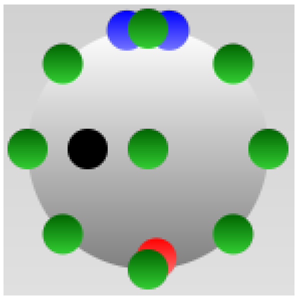


Figure 5: an example available placements on a circle FX node. The green circles have a corner* (or at least, what would have been the corner), edge* or middle placement, the blue ones an edge placement with -10 and 10 degrees, the black one a custom placement with relative position $(-10, 0)$ and the red one a random placement.

3.4 Commands

In the current version of GreenMirror, communication between the client and the server is one way: all supported commands are meant to be sent from the client to the server. With the information in this section and section 4.8, one can develop a completely new client or server that can work with the current version of GreenMirror. For a better overview, the commands have been divided into tables 2 and 3: the first is about commands relating to the visualizer and the handling of state-transitions, the second is about changes in the model. In the tables, the command name is in the upper left corner, the parameters, their type and their description on the right and the command description underneath those. This section is meant to give an overview of the currently available commands used within GreenMirror: it is not as detailed as the JavaDoc documentation that is available on the repository of this project.

Table 2: commands pertaining to the visualizer and the handling of state-transitions

Initialization	`width` The width of the visualizer window.	integer
	`height` The height of the visualizer window.	integer
	`defaultAnimationDuration` The default time animations will take to complete.	double
	`rotateRigidlyRelatedNodesRigidly` Whether the "A" node of a rigid relation should rotate rigidly when the "B" node is rotated.	boolean
The initialization command initializes and opens the visualizer with the passed parameters. This should come *before* any command pertaining to the model.		

SetAnimationDuration	`duration`	double	The duration of all following animations, in milliseconds.

This sets the duration of all atomic animations. For example: if the animation duration is set to 1000 milliseconds and five animations will be played sequentially, the total animation duration is 5000 milliseconds.

Flush	`delay`	double	The delay that is added after the previous animation, in milliseconds.

By default, all animations resulting from one state-transition are played parallel to each other. This command creates a new queue: the set of parallel animations created after this command are played after the set of previously created animations. Optionally, a delay can be added between the previous and upcoming set of animations. Also see section 4.7.

EndTransition

This command signals the server that the state-transition has ended.

StartVisualization

The command that tells the server that the visualizations may start. The current version of this server handles this by transitioning to the first state.

ExitVisualizer

This command communicates to the server that the visualizer should exit. In this version of GreenMirror, the client sends this command if a fatal error is encountered in the user's model.

Table 3: commands pertaining to the user's model

AddNode	`id`	integer	The unique, internal ID of the GreenMirror node.
	`identifier`	string	The identifier of the GreenMirror node: the user-defined type and name.

This signals that a node has been added to the user's model. The visualizer doesn't have to do anything yet: the FX is yet to be defined at this point.

AddRelation	`name`	string	The name of the relation.
	`nodeA`	integer	The internal ID of node A.
	`nodeB`	integer	The internal ID of node B.
	`placement`	string	The placement data of node A on node B.

	`rigid`	boolean	Whether the relation is rigid or not.
	`tempFX`	FxWrapper	The temporary FX of node A.

This indicates that a relation has been added between a node "A" and a node "B". If `placement` is set, the server should handle this. The same goes for `tempFX`.

RemoveNode	`id`	integer	The internal node ID.

This commands signals that a GreenMirror node has been removed from the user's model. Consequently, all relations have also been removed.

RemoveRelation	`id`	string	The unique ID of the relation.
	`nodeA`	integer	The internal ID of node A.

This commands signals that a relation has been removed. The server should also handle restoring the FX of node A in the case a temporary FX was set.

SetNodeFX	`id`	integer	The internal ID of the GreenMirror node.
	`fx`	FxWrapper	The FX values.

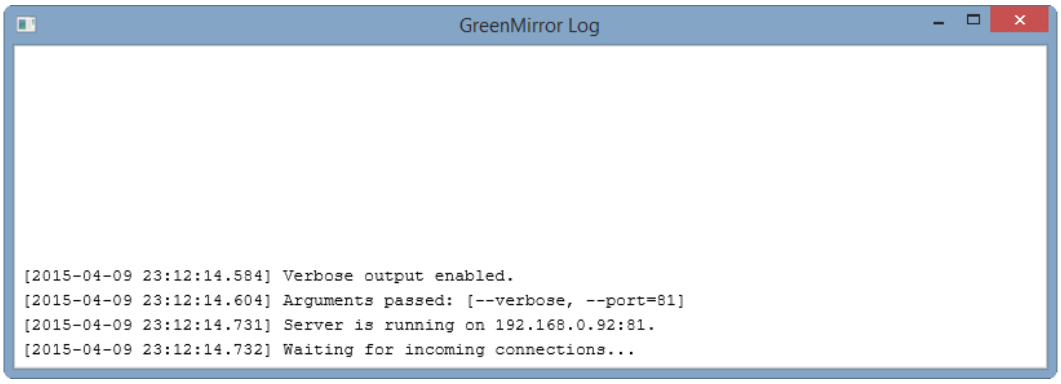
This commands communicates with what properties and values the FX of a node should be set. The `fx` parameter can include all properties that can be set, initially or otherwise (see section 3.2).

ChangeNodeFX	`id`	integer	The internal ID of the GreenMirror node.
	`fx`	FxWrapper	The new FX values.

This commands indicates that the FX of a GreenMirror node has been changed. The `fx` parameter can include only animatable properties (see section 3.2).

3.5 Log

It is assumed that any stakeholder working with the tool wants as much data and information as possible about what is happening. GreenMirror uses a static Log class that accepts any implementation of PrintStream. During runtime both the client and the server send data and information to their respective log sinks. A time stamp that is accurate to the millisecond is included with each entry, so performance can also be analysed. The client uses System.out as its log sink, seeing as it does not have a graphical user interface. The server uses System.out and, as it does have a graphical user interface, GreenMirror's WindowLogger implementation (depicted in figure 6). As was briefly mentioned in section 3.1, a verbose option can be enabled which results in the log being filled with more raw data.



```
[2015-04-09 23:12:14.584] Verbose output enabled.
[2015-04-09 23:12:14.604] Arguments passed: [--verbose, --port=81]
[2015-04-09 23:12:14.731] Server is running on 192.168.0.92:81.
[2015-04-09 23:12:14.732] Waiting for incoming connections...
```

Figure 6: an instance of WindowLogger right after the server has been executed

3.6 The GridBuilder class

GreenMirror has an auxiliary GridBuilder class created to take away the tedious work of building a grid of nodes. It supports properties such as the amount of cells, the cell width and height, cell spacing, cell colour, borders, a background colour and the type and name prefix for every GreenMirror node it creates. It builds the grid by creating one GreenMirror node per cell and one for the background, all of which have the rectangle FX type. Listing 3 shows an example of how short the code is to create a TicTacToe grid, in contrast to defining every single GreenMirror node, not to mention the tedious work of getting their exact positioning right. Doing the same without the GridBuilder class takes roughly 30 lines of sloppy code. The result of listing 3 is visible in figure 7.

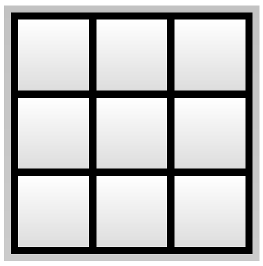


Figure 7: the result of the GridBuilder example code of listing 3

Listing 3: example code the user can use to build a grid of nodes

```
new GridBuilder("ticTacToeGrid:cell_")
    .setCellCount(3, 3)
    .setCellSize(50, 50)
    .setCellFill("linear-gradient(to bottom, #FFF, #DDD)")
    .setCellSpacing(5)
    .setBorderSize(5) // top, right, bottom and left
    .setBackgroundFill("black")
    .build(10, 10) // Coordinates on the visualizer
    .getNodes()
```

4 Design and implementation

This section gives more detailed information about the design of GreenMirror and some details of its implementation. GreenMirror is written with version 8 of the Java Runtime Environment and developed under Eclipse Luna. It highly depends on the integrated JavaFX library and uses JUnit [7] for testing, JOpt Simple [5] for handling the command line options (see section 4.3), Groovy [4] (see section 4.4) for its JSON implementation and scripting capabilities and the Eclipse JDT annotation package for using NonNull annotations. GreenMirror is composed of 114 classes in 12 packages, which sums up to a total of 7429 lines of code.

4.1 Package structure

GreenMirror's package structure is fairly self-evident. Still, a short explanation per package is in place. Details about how to extend certain subpackages are available in appendix A. Corresponding class diagrams depicting all relations between the classes are unfortunately too large to be a useful addition to this report, although they are available on the repository.

`greenmirror` is the main package containing classes shared by the client and server.

`greenmirror.client` contains all classes that pertain solely to the client.

`greenmirror.client.modelinitializers` contains all implemented model initializers that can be used by the client. See section 4.4.

`greenmirror.client.traceselectors` contains all implemented trace selectors that can be used by the client. See section 4.4.

`greenmirror.commandlineoptionhandlers` contains all command line option handlers, both for the client and the server. The @ClientSide and @ServerSide annotations indicate where they are used. See section 4.3.

`greenmirror.commands` contains all commands that are sent from the client to the server and vice versa. It also contains all handlers that interpret and handle received commands. The handlers have @ClientSide and @ServerSide annotations to indicate where their respective commands are received.

`greenmirror.fxpropertywrappers` contains all implemented wrappers for FX properties. See section 4.6.

`greenmirror.fxwrappers` contains all implemented FX wrappers. See sections 3.2 and 4.6.

`greenmirror.placements` contains all implemented placements. See section 3.3.

`greenmirror.server` contains all classes that pertain solely to the server.

`greenmirror.server.playbackstates` contains the playback states of the visualizer. See section 4.5.

`greenmirror.tests` contains several unit tests to validate the workings of GreenMirror. See section 5.

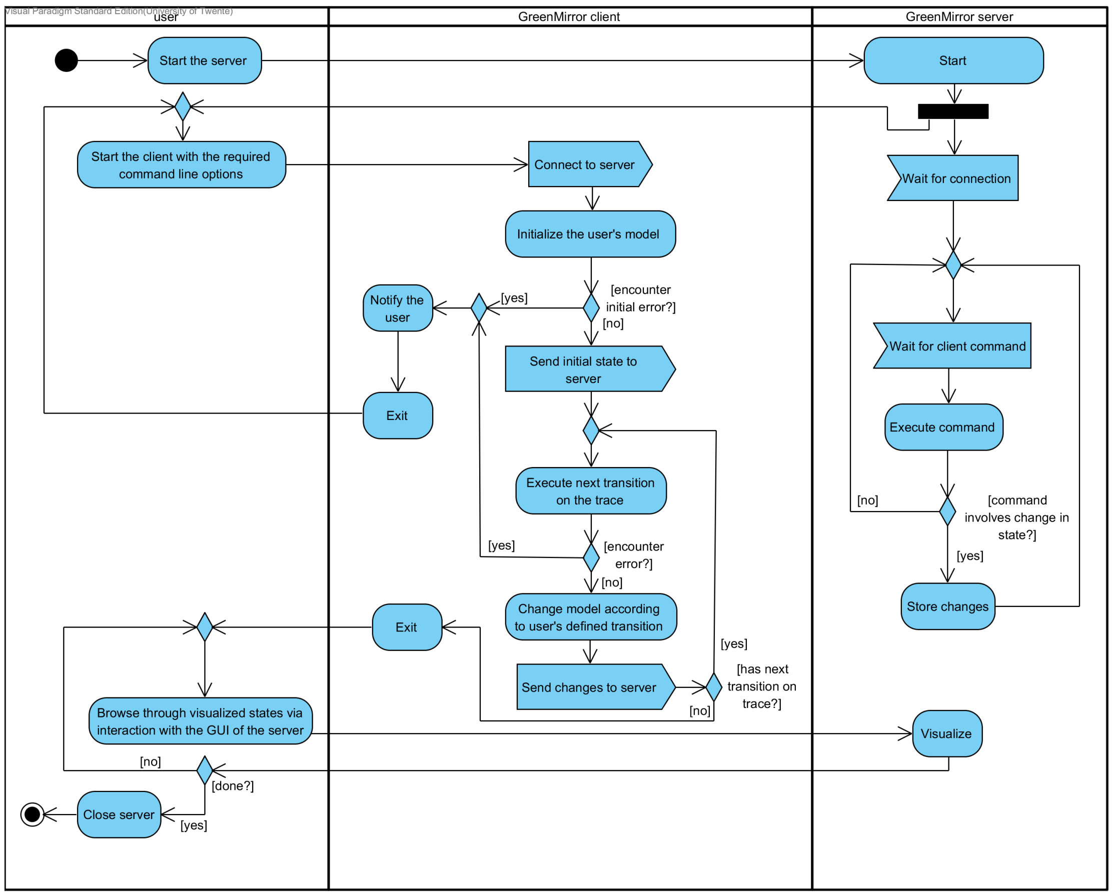

Visual Paradigm Standard Edition(University of Twente)
user
GreenMirror client
GreenMirror server
Start the server
Start
Start the client with the required command line options
Connect to server
Wait for connection
Initialize the user's model
[encounter initial error?]
[yes]
[no]
Notify the user
Send initial state to server
Exit
Wait for client command
Execute command
Execute next transition on the trace
[encounter error?]
[yes]
[no]
[no]
[command involves change in state?]
[yes]
Store changes
Change model according to user's defined transition
Exit
Send changes to server
[yes]
[has next transition on trace?]
[no]
Browse through visualized states via interaction with the GUI of the server
Visualize
[no]
[done?]
Close server
[yes]

Figure 8: simplified activity diagram of the general work-flow

4.2 General work-flow

The general work-flow of a typical execution of the application is illustrated in the simplified activity diagram of figure 8. This is meant to be fairly general: the exact work-flow depends on the used model initializers, the used trace selector and the user's model. In the current version, the visualization can only start when the whole model has been interpreted by the client and has been sent to the server. This behaviour is meant to ensure top performance while transitioning through the states, but can be easily modified.

4.3 Detailed work-flow

The first thing that occurs when operating the application is starting up the client or server and parsing the command line options. GreenMirror uses the JOpt Simple library [5] to parse command line options in the same way options can be used with executables of *nix operating systems. Available command line options implement the CommandLineOptionHandler interface. This contains everything needed to handle options: option and argument specification, processing order, argument validation and option processing. See figure 15 (appendix B) for the (simplified) sequence of these events. From the diagram can be seen that the options are all validated before they are processed. This prevents partial processing without having all required and valid options (for example: initializing the model without a valid server address).
The importance of the validating, parsing and processing of command line options, however,

should be explicitly stated. The complete set of the application's functions work as a direct consequence of the processing of these options. For example: the handler for the −−host option handles establishing the connection to the server and the handler for the −−trace option executes the trace. This results in the fact that new functionalities that should be executed during start-up can be easily added by implementing and adding new option handlers (see appendix A for instructions).

Next are the implemented command line option handlers of the client. From here on out, the assumption is made that all required options are passed (−−host, −−model and −−trace) and that their arguments are valid. If that would not be the case, GreenMirror would have observed this before processing and would have notified the user before terminating, as is described in the previous paragraph. The first option that will be processed is the −−host option, which connects to the server. This is very straightforward and will requires no further elaboration. See figure 9.

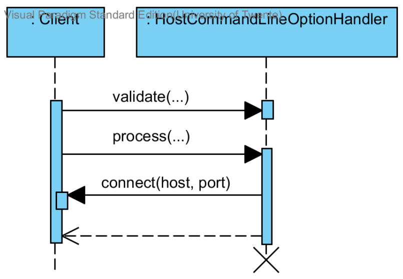


Figure 9: simplified sequence diagram of the handling of the −−host command line option

Handling the model initializer option −−model is far more interesting. Several notable things are worth stating when looking at the sequence diagram in figure 16 (appendix B). Firstly, it can be seen that multiple model initializers can be selected. This is designed this way so the user can define the model in more than one way, perhaps even with the use of modules. In practise this can be used by simply passing the −−model option multiple times. Multiple model initializers are executed in the same order as they were passed via the command line. Secondly, a model initializer should define the model with the initial state and the state-transitions. The initial state can be defined by defining the initial nodes and relations and adding them to the controller. This sends the information directly to the server. Each state-transition is defined as an instance of the ModelTransition class and holds a groovy.lang.Closure field. This is code that is executed when the transition is executed and should transition the model to the next state. How the model initializer exactly defines the initial state and the state-transitions is up to the implementation (see section 4.4). Finally, when the model initializers have been executed and thus the initial state has been defined, GreenMirror sends the StartVisualization command to the server, indicating that the transition to the first state can be performed.

The −−trace option is handled next. See figure 17 (appendix B) for the sequence diagram. This follows somewhat the same structure as the model initializer option handler, with a few slight differences. Only one TraceSelector can be used. However, multiple ModelTransitions can be executed with each transition from the trace, due to the fact that the ModelTransition instance has a regular expression pattern that matches with zero to unlimited transitions from the trace. These transitions are executed in the order in which the model initializer added them to the controller. After each executed transition, GreenMirror sends an EndTransition command to the server, indicating that a new state has been reached. If the user wants GreenMirror to refrain from sending this command, perhaps because the executed transition is part of the next one, the supplemental flag of the ModelInitializer instance can be set to true.

The client is now finished and will close. In the meanwhile, the server has received the commands the client has sent. Every command is passed to the correct CommandHandler in the sequence as shown in figure 10. What exactly happens in the handle(CommunicationFormat, String) method of the CommandHandler entirely depends on the received command.

After the client has sent the StartVisualization command, the server starts the transition to the first state. Upon finishing, the correct toolbar buttons are enabled and the user can start interacting with the visualizer. What is seen in the sequence diagram of the user interaction

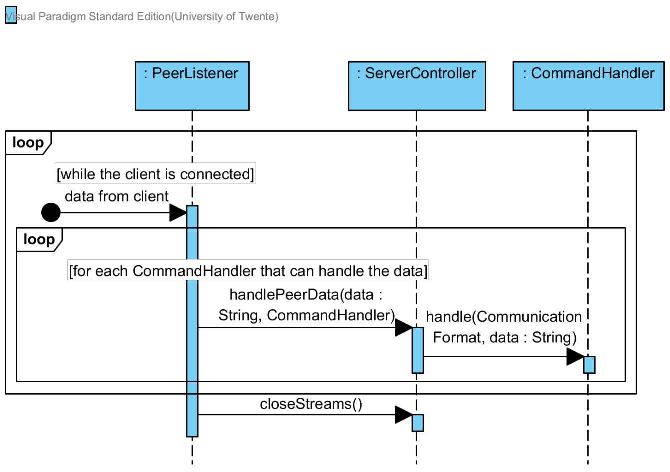

Visual Paradigm Standard Edition(University of Twente)
: PeerListener
: ServerController
: CommandHandler
loop
[while the client is connected]
data from client
loop
[for each CommandHandler that can handle the data]
handlePeerData(data : String, CommandHandler)
handle(Communication Format, data : String)
closeStreams()

Figure 10: simplified sequence diagram of the server receiving data

(figure 18 in appendix B) is that all visualization parameters of the state-transition are derived from the button the user clicked on.

4.4 Interface design

This section discusses the available interfaces for tool owners that provide ways of loading models into GreenMirror, and the current implementations. Loading a model consists of two parts: defining the model and providing the trace that defines in which order state-transitions will take place. Both parts have corresponding interfaces: respectively ModelInitializer and TraceSelector.

The model initializer has a few responsibilities. First and foremost it must, in the most general sense and not surprisingly, initialize the model according to the specifications of the user. More specifically: it must receive information from the user about how the initial state of the model is defined and how different state-transitions influence the model and the visualization. How the model initializer receives or retrieves this information is up to the implementation. Once it has received this information, it can add nodes to the client controller, remove nodes, add relations, etcetera. These changes are automatically conveyed to the server. The model initializer can also use the interface with the controller to send commands directly to the server, by use of the currently available commands in the greenmirror.commands package. This behaviour is, however, not recommended, because this circumvents the logic incorporated in updating the model via the controller. It is meant to provide the possibility to send auxiliary commands such as the SetAnimationDuration command.

The model initializers must define which state-transitions can happen by adding new instances of ModelTransition to the controller. As mentioned in section 4.3, this class has a groovy.lang.Closure field that changes the model when the closure is executed. This type is chosen to directly support the first implemented model initializer, although it is not restricted to this first implementation. The closure can accept arguments based on the transitions on the trace. This is best explained using an example. Suppose the user wants to visualize a ConnectFour game. It would be unwieldy to define a state-transition for every possible move (although there are only seven at the most), so the user defines one state-transition that uses the regular expression ^move([0−6])\$ to accept transitions from a trace. This means it needs the number of the column as an argument in the closure that executes the state-transition. Fortunately, the Groovy library supports this and GreenMirror takes advantage of this by supporting string and integer type arguments.

The server could be completely re-purposed by implementing different commands and command handlers. However, if the server is used as a visualizer, the model initializer has the responsibility of making sure the server first receives the initialization command. Without it, there is no JavaFX stage to which JavaFX nodes can be added. A tool owner developing a new model initializer could choose to delegate this responsibility to the user.

The model initializer that has been implemented in this first version of GreenMirror is based on Groovy [4]. Groovy is a dynamic language for the Java platform and provides a vast array of useful features. Specifically, the model initializer uses Groovy's script functionalities. This choice was made due to the following reasons.

1. The user receives the power and flexibility of a full-featured programming language, but still is easy to learn. This means that both advanced programmers and users without much experience can use it.
2. The user scripts can be executed during runtime, meaning that, while employing a complete programming language, the GreenMirror framework doesn't have to be recompiled every time the user alters his model.
3. A clear interface with the controller can be provided to the user, which Groovy calls a *base class*. The user can refer in his script to the base class' methods without referring to any object. This works as if the user is programming in the context of one of the methods of the base class (which it also comes down to, internally).

Listing 4 shows an example of a user script that can be executed by the Groovy script model initializer. Figure 11 shows a screenshot of its resulting visualization. It can be seen from the listing that chained statements are possible and actually encouraged to improve the readability of the script. Another notable advantage is that it is easily seen that calls to the base class (and thus indirectly to the controller) indicate a read from or a write to the model. For example, creating a new `Node` instance does not mean it is added to the model: `addNodes()` takes care of that.

Listing 4: example Groovy script defining the user's state-transition model

```
initialize(480, 200);
addNodes(
  new Node("loc:1").set(fx("rectangle")
                              .setSize(160, 200).setPosition(0, 0)
                              .setFill("radial-gradient(center␣␣80px␣100px,␣"
                                                          + "radius␣150px,␣white,␣red)")),
  new Node("loc:2").set(fx("rectangle")
                              .setSize(160, 200).setPosition(320, 0)
                              .setFill("radial-gradient(center␣400px␣100px,␣"
                                                          + "radius␣150px,␣white,␣red)")),
  new Node("obj")   .set(fx("circle")
                              .setRadius(20)
                              .setFill("linear-gradient(to␣bottom,␣limegreen,␣black)"))
);
addRelation(
  new Relation("on").setNodeA(node("obj"))
                         .setNodeB(node("loc:1"))
                         .setPlacement(Placement.MIDDLE)
);

addTransition("switch", {
  switchPlacementRelation(
      node("obj").getPlacementRelation().clone()
                                                .setNextNodeB(nodes("loc:"))
  );
});
```

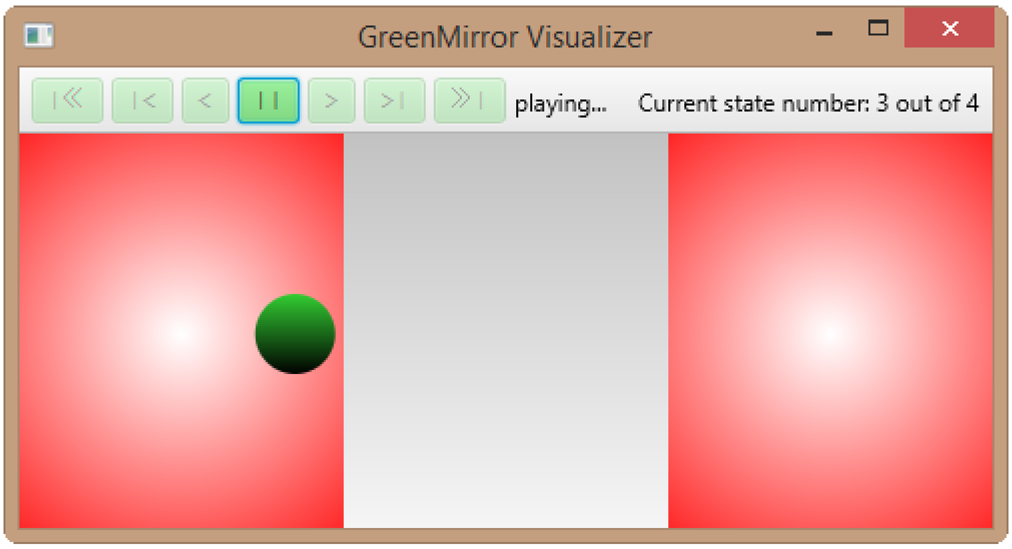


Figure 11: a screenshot of the executing visualization of listing 4 with the trace from listing 5

After the model initializer has set the initial state and saved all ModelTransitions, the selected TraceSelector is executed. The current FileTraceSelector implementation retrieves the trace from a text file where the transitions are separated by a newline. In the visualization example of figure 11, the trace file looked like listing 5. In the toolbar of figure 11 a state count of four can be seen, while three transitions are in the trace. This is naturally because the initial state is included in the count.

Listing 5: example trace file for the model defined in listing 4

```
switch
switch
switch
```

Both the ModelInitializer and TraceSelector interfaces accept one argument from the command line. This should be the source of the model and the trace, respectively. In GroovyScriptModelInitializer it is the file name of the Groovy script, while in FileTraceSelector it is the name of the file containing the trace. In future implementations that, for example, connect GreenMirror to another tool, this could be the name of the model.

4.5 Implemented design patterns

GreenMirror conforms to several design patterns [8, 11] to stimulate future development and the overall maintainability of the framework. Table 4 describes several implemented patterns, in alphabetical order. The name on the left side is the name of the design pattern. The right side describes the classes or structures that use the design pattern and contain information about the contexts in which the patterns are implemented.

Table 4: design patterns used in the GreenMirror framework

Builder	`GridBuilder` This class first accepts several required and optional parameters, after which it constructs the grid and finally returns the `NodeList` instance that contains the nodes that make up the grid. See section 3.6 for more information and an example.
Command	`Command` Whenever a subclass of `Command` is instantiated, the arguments are passed to its constructor. The `prepare()` method is called, in which the command can optionally execute preparations. Finally, the `getFormattedString(CommunicationFormat)` is called to retrieve a string that will be sent to the peer, formatted according to the parameter.
Memento	states and state-transitions The memento pattern is specifically designed to store and retrieve the internal state of an object. This object is, in the case of GreenMirror, the visualizer. The internal state data is composed of the collection of JavaFX nodes and their properties, and the JavaFX transition data to progress to a next state. This pattern is implemented with the `Visualizer` and `VisualizerMemento` classes. Needless to say, `VisualizerMemento` fulfils the memento rule. The `Visualizer` class fulfils both the caretaker and originator roles, implementing the `VisualizerMemento.Caretaker` and `VisualizerMemento.Originator` interfaces to make this more expressive. It should be noted that the current version of GreenMirror only has need to store the JavaFX transition data in the memento. More on this will be explained in section 4.7.
Model-view-controller	`Node`, `Relation` - `Visualizer`, `Log` - `GreenMirrorController` GreenMirror uses the MVC pattern to improve maintainability. The model is represented by `Node` instances. If the user changes the model, the `Client` instance (which extends `GreenMirrorController`) is notified and in turn notifies `Log` and the server. The view and the model have no interaction whatsoever on the client's side. On the server's side, however, the controller role is shared by the `Visualizer` and `ServerController` instances because of the integrated functionalities. The view on the server's side is implemented by both `Visualizer` and `Log`, although it can also be argued that the server as a whole represents the view.
Null object	`NullNode` Any GreenMirror node that gets removed in the user's model is replaced by an instance of `NullNode`. This makes sure the model throws expected exceptions and ensures the user will get properly notified if he tries to access it.

Observer	`Node`
	The `Client` controller is notified of model updates, but only of nodes that have been added to the visualizer.
	`FxWrapper`
	Every `Node` instance also is an observer: it observes any changes made in its FX.

Prototype	`FxWrapper`, `Placement` All implemented subclasses are instantiated using the built-in `java.util.ServiceLoader` injector. When a new instance of `FxWrapper` or `Placement` is requested based on a data string, the string is compared to the stored instances and if a match is found, a clone of the matched instance is returned. `Placement` implementations can also be instantiated using the `new` operator, but both `FxWrapper` and `Placement` are at some point constructed via the prototype design pattern.

Proxy	`FxWrapper`
	The `FxWrapper` class is exactly what the name says: a wrapper for the FX of a GreenMirror node. It generalizes handling the FX and provides intelligent access to certain FX properties.
	`FxPropertyWrapper`
	The `FxPropertyWrapper` implementations primarily provide intelligent access to several methods that are often used. It is added to simplify adding support for different FX types and properties.

State	`PlaybackState` The visualizer has a finite set of playback states it can be in. Certain things depend on the visualizer's playback state, such as which of toolbar buttons are enabled. The context role is taken up by the `Visualizer` instance, whereas the playback state is represented by instances of implementations of the `PlaybackState` class. Figure 12 shows the state machine diagram belonging to the playback states.
Strategy	`CommandHandler`, `CommandLineOptionHandler`, `FxWrapper`, `ModelInitializer`, `TraceSelector` All classes that implement the strategy design pattern do this so their subclasses can handle data in their own way. Which strategy (and thus which subclass of one of the above classes) is passed is determined at runtime. For example: every `ModelInitializer` implementation has its own `executeInitializer()` method that initializes the user's model in its own way. Which `ModelInitializer` is executed, depends on which the user selects during runtime. The `Log` class also uses this pattern, only it accepts `PrintStream` subclasses as strategies.

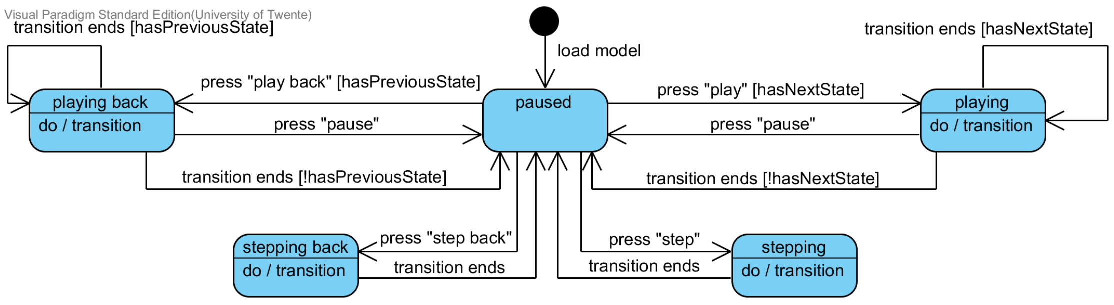

Visual Paradigm Standard Edition(University of Twente)
transition ends [hasPreviousState]
load model
transition ends [hasNextState]
press "play back" [hasPreviousState]
press "play" [hasNextState]
playing back
do / transition
paused
playing
do / transition
press "pause"
press "pause"
transition ends [!hasPreviousState]
transition ends [!hasNextState]
press "step back"
press "step"
stepping back
do / transition
transition ends
transition ends
stepping
do / transition

Figure 12: the state machine diagram for the visualizer's playback states

4.6 Internal representation of visual node properties

The FxWrapper class and its subclasses store and track the FX properties of GreenMirror nodes, as is explained in section 3.2. Using a wrapper instead of directly using a JavaFX node instance has the following reasons.

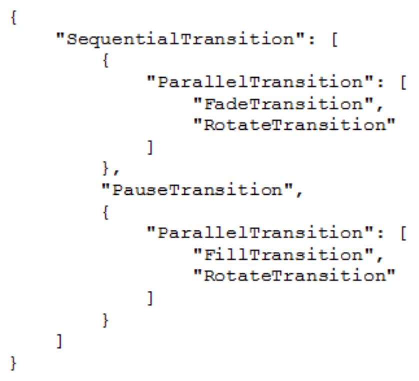
1. FxWrapper provides general methods such as converting FX data into an object that can be shared between client and server, or changing general properties like the rotation and opacity of a JavaFX node.
2. The implementations of abstract methods of FxWrapper used by GreenMirror might differ per type of JavaFX node. For example: the calculations of a specific placement on the edge of a circle differ from the calculations of a placement on the edge of a rectangle.
3. For the logic in the user's model and the proper creation of state-transitions on the server, properties have to be set and changed during the processing of state-transitions in the model without directly affecting the FX in the visualizer. As will be explained in section 4.7, changing actual properties of JavaFX nodes happens as a consequence of the execution of JavaFX transitions. To work with these values without directly visualizing them, a wrapping layer is needed providing 'virtual' values.
4. Support for new types of JavaFX nodes can be easily added. Examples include ellipses, three-dimensional shapes and composite nodes.

The way FxWrapper converts its properties to an object that can be sent over the network, is generalized in the sense that support for new types of properties can also be implemented easily. The following example will clarify this. One of the supported properties of ImageFxWrapper is the X coordinate of type double. In the current GreenMirror version, FX data is sent to the server in JSON format (see section 4.8). The value of the X coordinate can be easily converted to and from a string format, as would the boolean-type preserveRatio property. So how would the image property of type javafx.scene.image.Image be converted to and from a valid JSON string? The implementations of FxPropertyWrapper handle the FX properties to make this modular and easily extensible. In the case of the example about the image, the ImageFxProperty class handles the image property.

4.7 Internal representation of states and state-transitions

State data is not stored in the current version of GreenMirror, because it is unnecessary. To understand why this is, an explanation must be provided about how state-transition data is stored and how JavaFX handles its transitions (animations, in this case).

GreenMirror uses classes that extend javafx.animation.Transition. These classes hold all data needed to perform an animation: start and end value of an FX property and a method that handles the temporal behaviour of the value. JavaFX also provides two special transition implementations: ParallelTransition and SequentialTransition. These handle other transitions that are added to their lists in parallel or sequentially, respectively, and can be nested. This is also how GreenMirror stores a state-transition: one SequentialTransition instance holding one or multiple



```
{
    "SequentialTransition": [
        {
            "ParallelTransition": [
                "FadeTransition",
                "RotateTransition"
            ]
        },
        "PauseTransition",
        {
            "ParallelTransition": [
                "FillTransition",
                "RotateTransition"
            ]
        }
    ]
}
```

Figure 13: an example of one stored state-transition in JSON notation

ParallelTransition instances (separated by a PauseTransition to incorporate an optional delay) which in turn hold the individual Transition instances that animate the change in properties. Initially, every change in the model the server receives ends up in the same top-level ParallelTransition. The user might want to show several sequential animations during one state-transition. For this scenario, the flush command has been implemented. This results in the server creating a new ParallelTransition instance in the root SequentialTransition in which all further transitions will be stored. For an example of these nested transitions of one state-transition, see figure 13. JavaFX' transition classes also have another interesting functionality: the rate property. When set to a negative value, the animation reverses. This makes browsing backwards through the model's states very easy and removes the need to recalculate FX property values of JavaFX nodes.

These reasons make storing state data unnecessary. State-transition data is composed of the JavaFX animations needed to go from one state to another, including the start and end values of the relevant FX properties, whichever direction the user wants to go.

4.8 Interchange formats

Nothing is currently being stored of the result of a visualization. The formats associated with defining the user's model and the corresponding trace are discussed in section 4.4. This leaves the protocol used between the client and the server. This is very simple and has the form `command:commanddata`. The part before the colon holds the name of the command as it is presented in section 3.4. This is included so the receiver knows which CommandHandler implementation can handle the command data. The part after the colon holds the command data, formatted in the selected CommunicationFormat. The JSON format is currently the only supported format. An example of a command is shown in listing 6.

Listing 6: an example command sent from the client to the server in the JSON communication format, indicating that a node as been added to the model

```
AddNode:{"id":2,"identifier":"nodetype:nodename"}
```

5 Validation

Unit tests have been written with the JUnit library that validate the classes the user works with the most: `Node`, `Relation`, `NodeList`, `RelationList`, `FxWrapper`, `FxWrapper`'s subclasses, `Placement` and `Placement`'s subclasses. The unit tests include the various ways of instantiating new objects, calling their methods with extreme values and in the case of the `Placement` tests, they verify the calculations made to place a node in a certain respect to another node. These tests all worked as expected.

Due to the visual nature of GreenMirror, further verification is based on an extensive system test, written for use with the Groovy script model initializer. This test also works to some extent as a unit test: it tests several classes and specific functionalities. Furthermore, it showcases and explains how certain functionalities work and how they can be used. Among other sub-tests, it tests the following:

- setting the duration of animations;
- using parameters from state-transition names;
- adding nodes;
- setting the FX of nodes;
- altering the general properties of nodes;
- altering properties specific for shape nodes (rectangle, circle and text nodes);
- altering properties of a rectangle node;
- altering properties of an image node;
- altering properties of a text node;
- adding and removing simple relations between nodes;
- adding and replacing placement relations;
- using rigid placement relations;
- using chained, rigid placement relations; and
- removing nodes.

A coverage of 84.8% has been achieved with all tests combined. GreenMirror uses the `@NonNull` annotation with fields and method arguments and return types. Not all sub-packages of Java formally guarantee that they return a non-null object (although they do informally), resulting in several non-null checks that are effectively futile and practically non-reachable code.

All test cases discussed in section 1.4 have been successfully visualized. A screenshot of the results are illustrated in figures 4, 14a and 14b. They were all fairly easy to construct using the Groovy script model initializer. The realization of the ferryman case was a bit more elaborate compared to the realization discussed in section 2, although the case itself and figure 4 are essentially the same for both realizations. The test case, for example, also includes the state-transition where one cargo object 'eats' another. Table 5 lists all relevant results of the three test cases.

Table 5: test case results

Test case	**lines of code**	**trace length**	**init. time**[2]	**notable visualizations and used functionalities**
Ferryman	109	20	≈ 48 s	Parametrized state-transition names, rectangle FX types, image FX types, node resizing, node rotation, node removal, placement relations, rigid placement relations, non-placement relations, relation replacements, colour gradients
ConnectFour	91	10	≈ 8 s	Parametrized state-transition names, rectangle FX types, circle FX types, text FX types, node grid generated with `GridBuilder`, node creation during state-transition, colour gradients, font size setting, placement relations, relation replacements, complex model logic (determining the cell number from the column number), model failing (if an invalid move is encountered on the trace), altering animation duration
Dining Philosophers	132	15	≈ 10 s	Parametrized state-transition names, circle FX types, image FX types, image alteration during state-transition, node rotation, placement relations, non-placement relations, relation replacement, model failing (if an impossible state-transition is encountered)

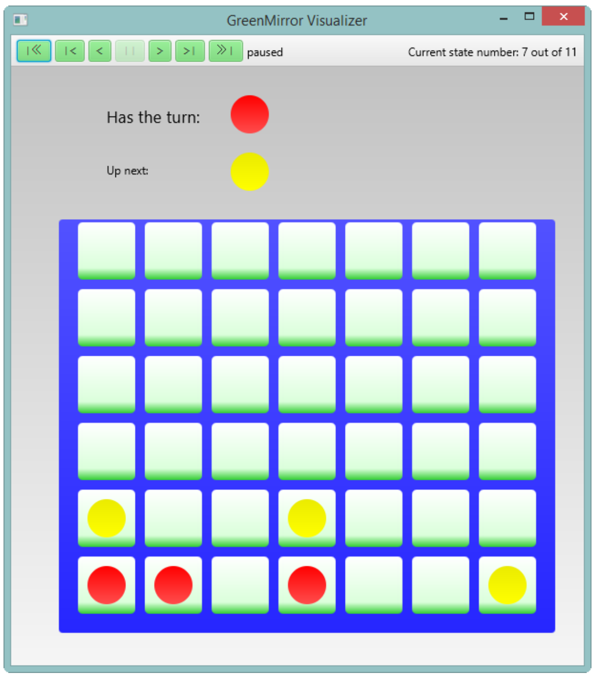
The following list discusses how each requirement and use case is implemented and validated, in the same order as they were defined in section 1.4.

Requirement 1 and use case 1
The framework is easily extensible due to the use of interfaces and abstract classes. One example is the FxWrapper class: support for new FX elements can be easily added by extending FxWrapper. Also, the use of common design patterns makes extension more easy. See appendix A for a full list of extensible parts.

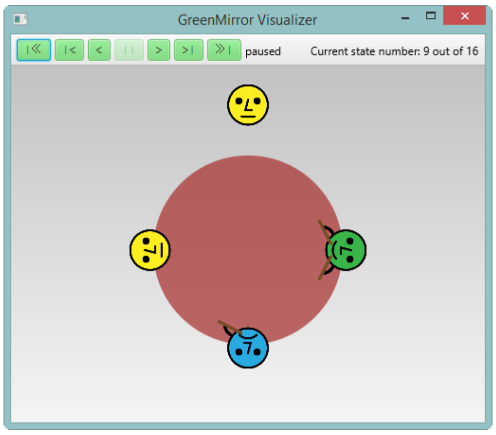
Requirement 2
The use of common design patterns, extensive documentation and the use of the checkstyle plugin of Eclipse are the primary means of making the framework maintainable. Especially the model-view-controller pattern is important because it tells which classes are responsible for what. More details on the design patterns can be found in section 4.5.

Requirement 3 and use cases 2 and 3
The application becomes aware of the user's model via the use of the model initializers: implementations of ModelInitializer. It depends on the implementation how this is done. The choice of the implementation is how use case 2 is supported: by use of the command line options and specifically the ModelCommandLineOptionHandler class that handles the choice. This option handler accepts one argument: the source of the model (use case 3) and it

[2]Initialization time: the time GreenMirror needs to interpret the model and generate the visualizations.



(a) ConnectFour



(b) Dining Philosophers

Figure 14: screenshots of two of the test cases

depends on the ModelInitializer implementation how this source is given (for example: a file name or a model name).

Requirement 4 and use case 4
The user can pass his choice for the trace source via the command line and gets handled by the TraceCommandLineOptionHandler class. The user's chosen TraceSelector implementation then retrieves the trace and that is how the application becomes aware of the trace.

Requirements 5 to 10 and use case 5
A few examples of where the validation of these requirements is visible (as seen in figures 4, 14a and 14b): the ConnectFour test case visualizes rectangles, circles (requirement 6) and text (requirement 7); the ferryman test case visualizes images (requirement 8); and the dining philosophers test case visualizes the placement of nodes with respect to other nodes (requirement 10). All test cases visualize simple animations (requirement 9), which of course is not visible in the figures. The fact that these test cases can be visualized validates requirement 5 and use case 5.

Requirement 11 and use case 6
This is implemented by the Log class, as is discussed in section 3.5 and visible in figure 6 (page 18).

Requirement 12 and use case 7
Browsing from state to state is implemented by use of a toolbar with navigation buttons. The workings of these buttons are handled by the Visualizer controller in cooperation with the PlaybackState implementations and ToolbarButton class. Section 4.5 discusses the playback state and its effect on the toolbar buttons. The buttons are also visible on any of the screenshots of GreenMirror in this report.

Requirements 13 and 14
Smoothly transitioning from state to state (requirement 13) is implemented by cleaning

up resources every time the visualizer is closed. This 'resets' the server so a new client can connect. Crashing is prevented (requirement 14) by properly catching exceptions while the model is being loaded: this makes sure the visualizer will not crash during the state-transitions. If any exceptions are thrown, they will be displayed in the log.

6 Discussion

GreenMirror is the second step in the development of an extensive research tool. The work of Aalbertsberg [1] was the first step, although except for minor design choices such as the programming language and the client-server structure, his work bears virtually no resemblance to the current version.

The intention for using a proper framework design was present during the design phase. Examples include the work of Fayad & Schmidt [3] and Markiewicz & De Lucena [9], but this has crept unwittingly into the background during the implementation phase. Some concepts have been used (e.g. hot spots), but more research should be used while further developing the GreenMirror framework. A somewhat similar point applies to the use of the MVC pattern. It is implemented, as discussed in section 4.5, but there is room for improvement to increase the framework's maintainability. For example: the distinction between the controller and view roles on the server side could be made more apparent and the view role on the client side could be improved beyond the use of just the Log class.

Help from work relating to GreenMirror's way of visualization could not be uncovered. Due to the absence of related work backing the used visualization approach, this is a point of discussion. "Visualization approach" here means: the way of visualization (nodes and their FX representation) and the internal representation, which are discussed in sections 3.2, 4.6 and 4.7. I believe the current approach is optimal for the requirements set for this project. It makes sure state-transitions can take place smoothly and without delay. It would probably, however, not be sufficiently efficient when GreenMirror is developed beyond the scope of this project. As is discussed in section 5, it can take GreenMirror nearly a minute to convert a simple model to a visualization. That model had merely 20 state-transitions. Therefore, my first recommendation is to evaluate the current programmed structures and internal representations.

There is currently a TraceSelector implementation in development by the Formal Methods and Tools research group of the University of Twente to select a trace from the GROOVE application. A next improvement to GreenMirror could be the development of a ModelInitializer implementation that can load a model from an existing GROOVE Grammar. This would narrow the bridge between the two tools and would certainly be considered a useful functionality. The user might be required to provide extra information about how GROOVE's nodes should be represented on the visualizer, should such an implementation be developed. Fortunately this can be done rather simple: the user can provide a script that uses the Groovy script model initializer to supplement the model, which is possible because the use of multiple model initializers is supported.

To conclude this report, some thoughts must be given about the ultimate goal of this tool. In addition to generating visualizations from a defined model, the ultimate goal is, as is briefly mentioned in section 1.4, generating a model definition from of user interaction with visualizations.

Take a model where a state-transition results in the movement of a node from one location to another. A next extension of GreenMirror could allow the user to drag a node in the visualizer from one location to another. Assuming both location boundaries have been properly defined, the application can recognize this as a state-transition. In this scenario the user can alter the trace, adding state-transitions before, in-between or after the transitions on the original trace.

There are of course many more visualization possibilities than simply moving a node. Continuing and expanding on the previous scenario: in stead of the atomic interaction of dragging a node to another location, the user could record multiple changes in the visualizer, resulting in a new state-transition definition or in the recognition of a previously defined state-transition. This might work fine solely for recognizing state-transitions, but this still has considerable limitations in the creation of new state-transitions.

The next step is the addition of the creation of model logic. In the GreenMirror application, a transition in the model could be defined as such: "if node A and node B both have a relation with node C, remove the relation between node A and node C and add a new relation between node A and node D". In this step of the development of this extension, a user should be able to create such logic based on his interactions with the visualizer. This can become complex very fast, but that should be considered an interesting challenge to accept in the future.

The next and perhaps final step is to make a two-way connection with other tools that enable or facilitate the research of state-transition models in different ways. When the connection is made between interactions with the visualizer and the creation of state-transition logic, this could be translated into a format that can be accepted by other tools. This way, the need is eliminated for researchers to rewrite their models into tool-specific formats when different tools are used.

References

[1] A. Aalbertsberg. Dynamic visualization of state transition systems. https://github.com/Vaeil/StateTransAnimation/blob/master/report/report.pdf, 2015.

[2] E.W. Dijkstra. Co-operating sequential processes. *F. Genuys (ed.): Programming Languages*, pages 43–112, 1968.

[3] Mohamed Fayad and Douglas C. Schmidt. Object-oriented application frameworks. *Commun. ACM*, 40(10):32–38, October 1997.

[4] Groovy: A multi-faceted language for the java platform. http://groovy-lang.org/.

[5] P. Holser. JOpt simple: A java library for parsing command line options. https://pholser.github.io/jopt-simple/.

[6] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2004.

[7] Junit: a simple framework to write repeatable tests. http://junit.org/.

[8] P. Kuchana. *Software architecture design patterns in Java*. CRC Press, 2004.

[9] Marcus Eduardo Markiewicz and Carlos J. P. de Lucena. Object oriented framework development. *Crossroads*, 7(4):3–9, July 2001.

[10] A. Rensink. The GROOVE simulator: A tool for state space generation. *Applications of Graph Transformations with Industrial Relevance*, pages 479–485, 2004.

[11] A. Shvets. *Design Patterns Explained Simply*. Sourcemaking.com.

A Service extension instructions

This appendix gives instructions on how to extend several components of the GreenMirror framework. These components have been specifically designed to be easily extensible with as few steps as possible. The final, omitted step after adding an extension is to recompile, so the extension can be used. This section is meant for developers, although appendices A.1 and A.2 are mainly meant for tool owners. Both of these stakeholder groups are assumed to have sufficient understanding of the Java programming language to comprehend these instructions. Most extensible components make use of the `java.util.ServiceLoader` injector. More in-depth details about implementing new code is available in the JavaDocs on the repository of this project.

A.1 `ModelInitializer`

1. Implement `greenmirror.client.ModelInitializer`, making sure that it has a zero-argument constructor.
2. Add the fully-qualified binary class name of your new class with a new line to the file `META-INF/services/greenmirror.client.ModelInitializer`.

A.2 `TraceSelector`

1. Implement `greenmirror.client.TraceSelector`, making sure that it has a zero-argument constructor.
2. Add the fully-qualified binary class name of your new class with a new line to the file `META-INF/services/greenmirror.client.TraceSelector`.

A.3 `FxWrapper`

1. Extend `greenmirror.FxWrapper` or `greenmirror.FxShapeWrapper` if the JavaFX node type your new class is representing is an extension of `javafx.scene.shape.Shape`. In either case make sure that it has a zero-argument constructor.
2. Add the JavaFX node properties you want to support (see appendix A.4).
3. Add the fully-qualified binary class name of your new class with a new line to the file `META-INF/services/greenmirror.FxWrapper`.

A.4 `FxPropertyWrapper`

1. Extend `greenmirror.FxPropertyWrapper`.
2. Support for this FX property type is being added to support a specific FX property of an `FxWrapper` subclass. An entry must be added to one of two methods of the relevant `FxWrapper` subclass. If the FX property can be animated, add an entry to the `getAnimatableProperties()` method. If the property can only be set once, add it to the `getChangableProperties()` method.
3. Add one or more get-methods to the `FxWrapper` subclass.

4. Add one or more set-methods to the FxWrapper subclass. The type of the argument of the primary set-method depends on what the relevant FxPropertyWrapper's getPropertyType() method returns.

5. If the property can be animated, but hasn't got a javafx.animate.Transition implementation yet, create it. The abstract AbstractTransition and DoublePropertyTransition classes have been created to provide several often used methods when extending javafx.animate.Transition.

6. If the property can be animated, add the animate method that returns the javafx.animate.Transition that changes the value of the property when played.

7. If the user needs access to the new property type in the Groovy model initializer, add an entry to the IMPORTS constant of the greenmirror.client.modelinitializers.GroovyScriptModelInitializer class.

A.5 Placement

1. Extend greenmirror.Placement, making sure that it has a zero-argument constructor.

2. If the placement has no further parameters (such as the angle parameter of EdgePlacement), add a public constant to greenmirror.Placement holding an instance of your new class.

3. Add support for the new placement by adding the necessary calculations to the calculatePoint(Placement) methods of all implemented FxWrappers and to the static calculatePointOnRectangle(double, double, Placement) method of the FxWrapper class.

4. Add the fully-qualified binary class name of your new class with a new line to the file META-INF/services/greenmirror.Placement.

A.6 Command

1. Extend greenmirror.Command.

2. Add a corresponding CommandHandler. See appendix A.7.

A.7 CommandHandler

1. Extend greenmirror.CommandHandler, making sure that it has a zero-argument constructor and that its getCommand() method returns the same string as the Command class that this handler is meant to handle.

2. Add at least one of the @ClientSide and @ServerSide annotations, indicating on which "side" the command should be handled.

3. Add the fully-qualified binary class name of your new class with a new line to the file META-INF/services/greenmirror.CommandHandler.

A.8 CommandLineOptionHandler

1. Implement greenmirror.CommandLineOptionHandler, making sure that it has a zero-argument constructor.

2. Add at least one of the @ClientSide and @ServerSide annotations, indicating on which "side" the command line option should become available.
3. Add the fully-qualified binary class name of your new class with a new line to the file META−INF/services/greenmirror.CommandLineOptionHandler.

A.9 Log

1. Extend java.io.PrintStream.
2. Add a Log.addOutput(instance); statement to the entry point of the component you want to add it to. The entry point of the client is the static main(String[]) method of the greenmirror.client.Client class. The entry point of the server is the start(Stage) method of the greenmirror.server.Visualizer class.

B Sequence diagrams

All diagrams are simplified in the sense that they do not show every atomic operation and that they show only validation and error handling when it is relevant and essential to the understanding of the sequences. They are simplified to improve the overall orderliness and comprehensibility of the diagrams.

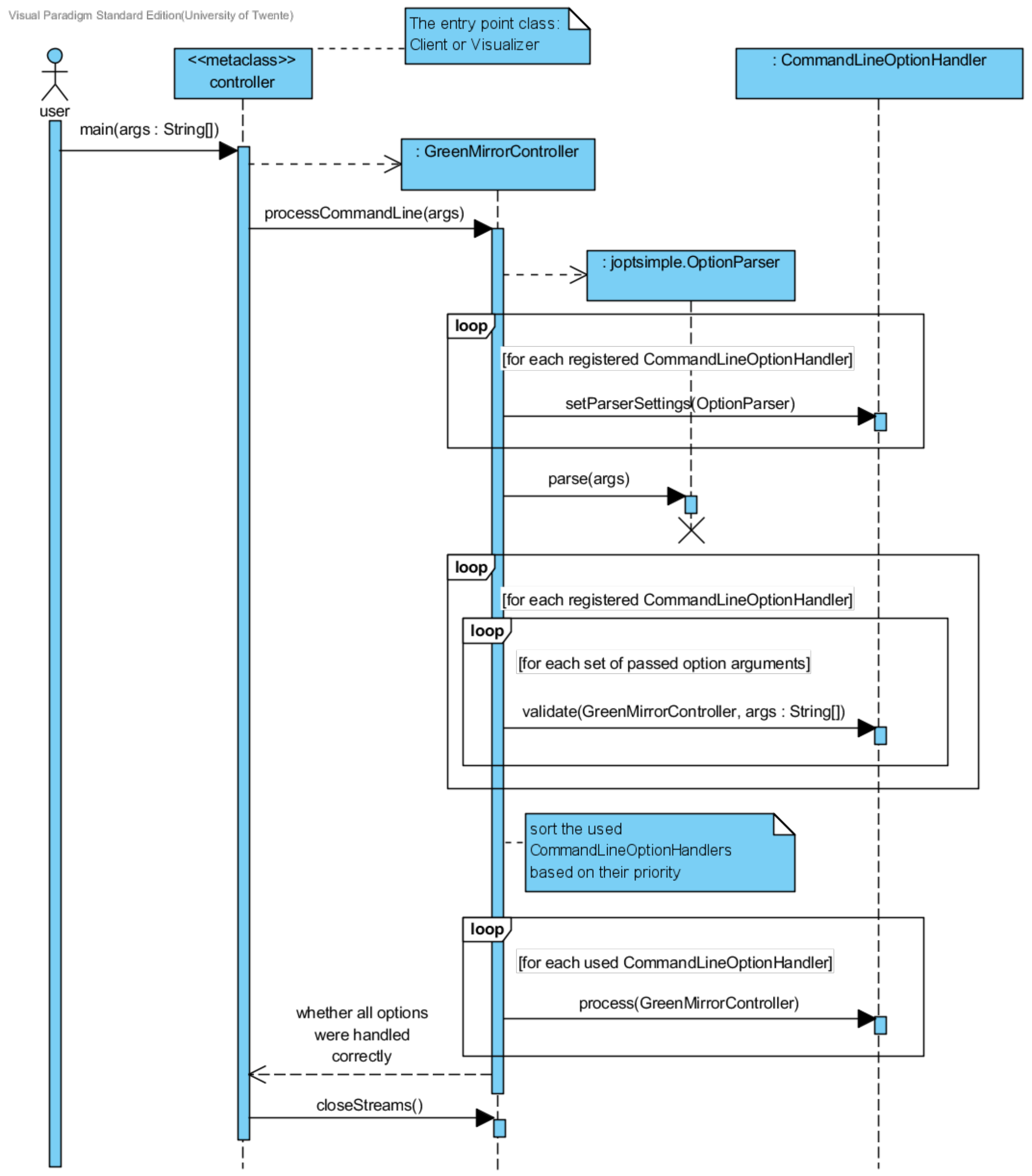
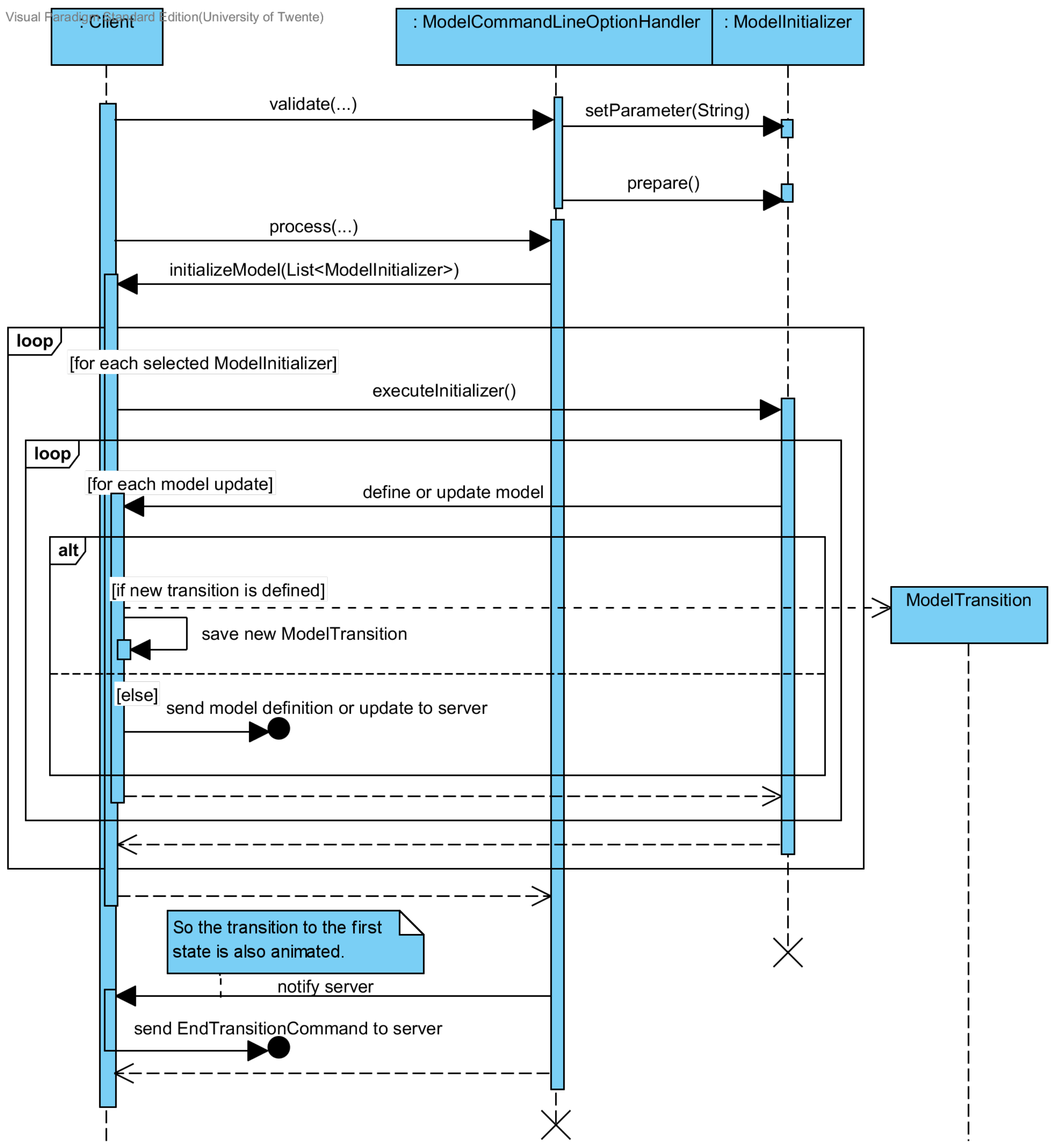

Visual Paradigm Standard Edition(University of Twente)
The entry point class: Client or Visualizer
<<metaclass>> controller
: CommandLineOptionHandler
user
main(args : String[])
: GreenMirrorController
processCommandLine(args)
: joptsimple.OptionParser
loop
[for each registered CommandLineOptionHandler]
setParserSettings(OptionParser)
parse(args)
loop
[for each registered CommandLineOptionHandler]
loop
[for each set of passed option arguments]
validate(GreenMirrorController, args : String[])
sort the used CommandLineOptionHandlers based on their priority
loop
[for each used CommandLineOptionHandler]
process(GreenMirrorController)
whether all options were handled correctly
closeStreams()

Figure 15: simplified sequence diagram of the general start-up. There is a slight difference on the server side: if the options are all handled correctly, the controller doesn't close the streams, but starts listening for incoming connections.



Visual Paradigm Standard Edition(University of Twente)
: Client
: ModelCommandLineOptionHandler
: ModelInitializer
validate(...)
setParameter(String)
prepare()
process(...)
initializeModel(List<ModelInitializer>)
loop
[for each selected ModelInitializer]
executeInitializer()
loop
[for each model update]
define or update model
alt
[if new transition is defined]
ModelTransition
save new ModelTransition
[else]
send model definition or update to server
So the transition to the first state is also animated.
notify server
send EndTransitionCommand to server

Figure 16: simplified sequence diagram of the handling of the −−model command line option

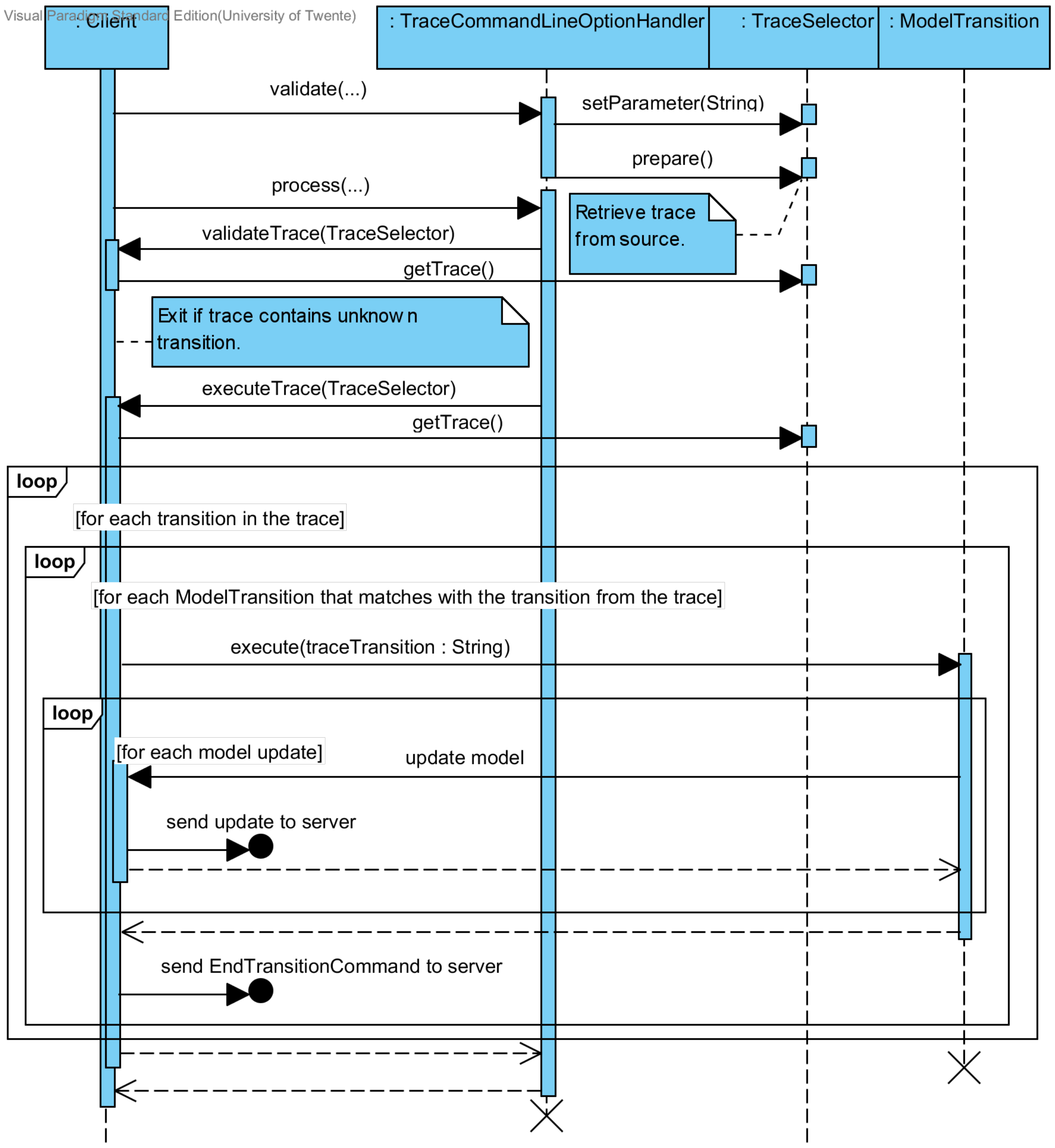
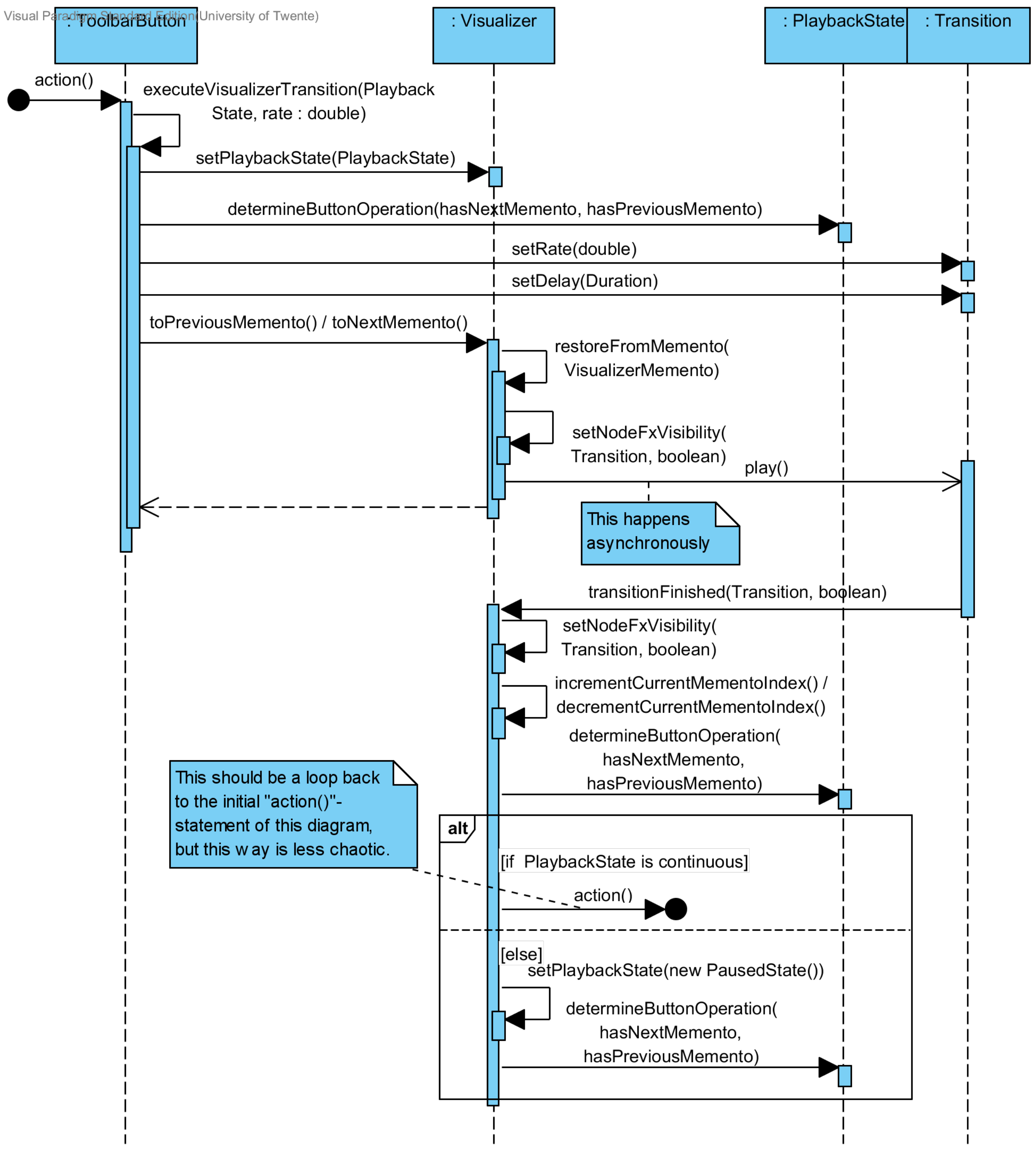

Visual Paradigm Standard Edition(University of Twente)
: Client
: TraceCommandLineOptionHandler
: TraceSelector
: ModelTransition
validate(...)
setParameter(String)
prepare()
process(...)
Retrieve trace from source.
validateTrace(TraceSelector)
getTrace()
Exit if trace contains unknown transition.
executeTrace(TraceSelector)
getTrace()
loop
[for each transition in the trace]
loop
[for each ModelTransition that matches with the transition from the trace]
execute(traceTransition : String)
loop
[for each model update]
update model
send update to server
send EndTransitionCommand to server

Figure 17: simplified sequence diagram of the handling of the −−trace command line option



Visual Paradigm Standard Edition(University of Twente)
: ToolbarButton
: Visualizer
: PlaybackState
: Transition
action()
executeVisualizerTransition(Playback State, rate : double)
setPlaybackState(PlaybackState)
determineButtonOperation(hasNextMemento, hasPreviousMemento)
setRate(double)
setDelay(Duration)
toPreviousMemento() / toNextMemento()
restoreFromMemento(VisualizerMemento)
setNodeFxVisibility(Transition, boolean)
play()
This happens asynchronously
transitionFinished(Transition, boolean)
setNodeFxVisibility(Transition, boolean)
incrementCurrentMementoIndex() / decrementCurrentMementoIndex()
determineButtonOperation(hasNextMemento, hasPreviousMemento)
This should be a loop back to the initial "action()"-statement of this diagram, but this w ay is less chaotic.
alt
[if PlaybackState is continuous]
action()
[else]
setPlaybackState(new PausedState())
determineButtonOperation(hasNextMemento, hasPreviousMemento)

Figure 18: simplified sequence diagram of user interaction with any one of the toolbar buttons (but not with the pause button)