GreenMirror: A Visualization Framework for State-Transition Models

Karim El Assal
s1097539

June 12, 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 framework using the Java programming language and the JavaFX library is being developed to provide a different approach. Continuing previous work, the next step has now been made in the development, resulting in a version 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, extendible and maintainable.

Contents

1 Introduction

1.1 Background

State-transition models describe the discrete dynamic behaviour of a system. Such a model defines how the state of a system changes as the result of a trigger, and following its set of rules. In the field of computer science, these models are used for system verification and they can be analysed in several ways. One often used way is by means of a *state space generation* tool such as GROOVE (see [7]).

The state spaces generated by such tools 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 based on a 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 (see [2]). This report describes which further progress has been made since then to reach the current version that has been named *GreenMirror*. Although the project is still in its infancy, hopefully these contributions will eventually lead to better state-transition model research.

1.2 Glossary

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 have a short designation to indicate that this project is meant in certain contexts.

JavaFX transition The animation of a JavaFX node property from one value to another.

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

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

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 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.

2 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 have already been visualized in a different way in state space tools, as described in section 1.1. The framework should be easily extendible and modifiable, so it can be improved in the future. Several requirements are described in the following subsections to properly define the requisites that will be used in the design and development phases. All requirements are *MoSCoW* formulated.

2.1 Architectural requirements

In the architectural requirements of the framework, the developer is the main stakeholder since he has most to gain from a well-defined software system. These requirements are fairly general and vague, but have to be stated nevertheless.

Extendibility The framework must be easily extendible. Future research might require new functionalities, so a developer or tool owner must be able to extend in stead of heavily modify the source code.

Maintainability 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.

From these requirements, one could extract the implicit requirement of modularity. Although it is not defined as a fixed requirement, it will be taken into account.

2.2 Functional requirements

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 requirement is the basis for all following requirements, which are mostly described in the order of usage by the user.

- **The application must become aware of the researched model** and how to visualize it. This process is divided into several parts.
  - 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.
- **The application must become aware of the trace**. Executing this trace gives the user the opportunity to analyse the model's dynamic behaviour.
- The application must **show the visualization to the user** generated from the user's specified model. As a basis for this requirement, the application must be able to visualize simple shapes (such as rectangles, circles, text and images), simple animations (such as moving, creating and removing nodes) and the placement of one node in a specific respect to another. See section 2.5 for specifics.
- The application should **provide a detailed log** about what it has done. This helps in debugging and improving the application and in analysing the researched model.
- As a result of the user's interactions, the application should be able to **move back and forth between the transitions** given by the trace 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.

2.3 Performance requirements

The performance requirements also relate to the application and primarily concern the smooth and uninterrupted execution of visualizations.

- After the model has been loaded into GreenMirror, the visualizer should be able to **transition to a next or previous state without noticeable delay** caused by memory or processing problems.
- While transitioning to a next or previous state, **the application should not crash** or terminate otherwise. This includes the assumption that the model has successfully been loaded into GreenMirror and all model logic has been deemed valid and without errors.

2.4 Possible extensions

The scope of these contributions to the project are limited. Had it been larger, an extension that could have been implemented is the reverse of the current main goal of GreenMirror. In

addition to visualizing a defined model, a model could be altered or even defined from scratch based on how the user interacts with the visualizer.
This can be explained by introducing the following, relatively simple scenario. Take a model where a state-transition results in the moving of a node from one location to another. With the proposed extension implemented, the user might now be able 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. Amongst others, on which this report will elaborate more later, relations between nodes can be created and removed, nodes can change colour and size, new nodes can be added and existing nodes can be removed. This could also be implemented in an extension. 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 one newly defined or previously defined, recognized state-transition. This might work fine solely for recognizing simple, previously defined state-transitions, but this still has considerable limitations in the creation of new state-transitions.

Another step that should be taken in the development of this extension to enable the user to create a model based on interactions with the visualizer, 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 is just an interesting challenge to accept in the future.

The next 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 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.

2.5 Requirements validation

Several well-known test models have been defined to validate the functional and performance requirements. These models are not extremely complex, but they do depict the possibilities of the application and show that the requirements are met. All models described below validate the following functional requirements by simply showing the visualization. If one of these requirements is not met, the model could not have been visualized.

- The application must become aware of the researched model.
- The application must become aware of the trace.
- The application must show the visualization to the user.

While viewing these models, sufficient data and information about what is happening should show in the log, as described by the requirement "show a detailed log". "Sufficient" in this context means in addition to all exceptions, everything that changes in the visualizer: adding and removing nodes and relations and changing visual properties of nodes.

2.5.1 The ferryman

This 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. 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.

Possible transitions could be: load_wolf, load_goat, load_cabbage, cross, unload and eat. The application must at least be able to visualize

- the banks of the river as two coloured rectangles;
- the wolf, the goat, the cabbage and the ferry as images;
- the loading of the passengers as their respective images moving from the bank they are on to the ferry;
- the crossing of the ferry with its (optional) passenger as their respective images moving to the other bank;
- the unloading of a passenger as its image moving from the ferry to the bank the ferry is moored to;
- the eating of the goat or the cabbage by letting the correct image disappear.

2.5.2 ConnectFour

The game ConnectFour is such a well-known game that the rules will be left out of this report. The initial state of the system is an empty seven-by-six board and two players with differently coloured tokens. The application must at least be able to visualize

- the players' token colours;
- the player whose turn it is and who is next, in textual form;
- the making of a move as a circular token falling through the specified column;
- the switching of the player whose turn it is.

2.5.3 The Dining Philosophers

E.W. Dijkstra came up with the Dining Philosophers problem (see [3]), 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 don't 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.

The GreenMirror application should be able to handle this model. It should at least show

- philosophers around a circular table, as images of simple smiley faces as seen in [6];
- the picking up and releasing of a fork, as the change of the philosopher's image;
- and most importantly: the logic that comes with the state-transitions, bearing in mind that the actual order of state-transitions is determined by the trace.

3 General design

Nearly all design choices have been made bearing in mind that extendibility is one of the most important requirements. The first large, architectural choice that influences all other choices is the separation of the client and server roles. The client in this project is the component that interprets the user's model, whereas the server is the component that performs the actual visualization as a consequence of the changes in the model. With this separation, the two distinct components can be linked to other components or to other component versions in the future.

Tools such as GROOVE 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 (more on this in section 3.4), 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.

This might be overstating what has already been mentioned in section 1.2, but in this project the term "node" is used two types of nodes: for a node in the model: a GreenMirror node; and a node of JavaFX: a (Java)FX node. When speaking simply of a node, a GreenMirror node is intended. In any other case, the words "JavaFX node" will be used. Solely using "FX" refers to the visual appearance of a GreenMirror node.

Different extendible parts of GreenMirror are loaded using the ServiceLoader class, a dependency injector implementation of Java. This enables an easy and well-known way for adding new implementations and keeping an overview on which classes are implementations of which part.

3.1 Package structure

GreenMirror's package structure is fairly self-evident. Still, a short explanation per package is in place. The sub-packages that contain implementations of extendible features will be elaborated on in section 3.4.

`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.

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

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

`greenmirror.commands` contains all network commands that are sent from the client to the server or vice versa. The @ClientSide and @ServerSide annotations indicate where from where they are sent.

`greenmirror.fxpropertywrappers` contains all implemented wrappers for FX properties.

`greenmirror.fxwrappers` contains all implemented FX wrappers.

`greenmirror.placements` contains all implemented placements.

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

`greenmirror.server.playbackstates` contains the playback states of the visualizer.

`greenmirror.tests` contains several unit and system tests to validate the workings of GreenMirror.

3.2 General work-flow

The general work-flow of a typical execution of the application is illustrated in the simplified activity diagram of figure 1. 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. However, one note is worth mentioning. In the current design, 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.

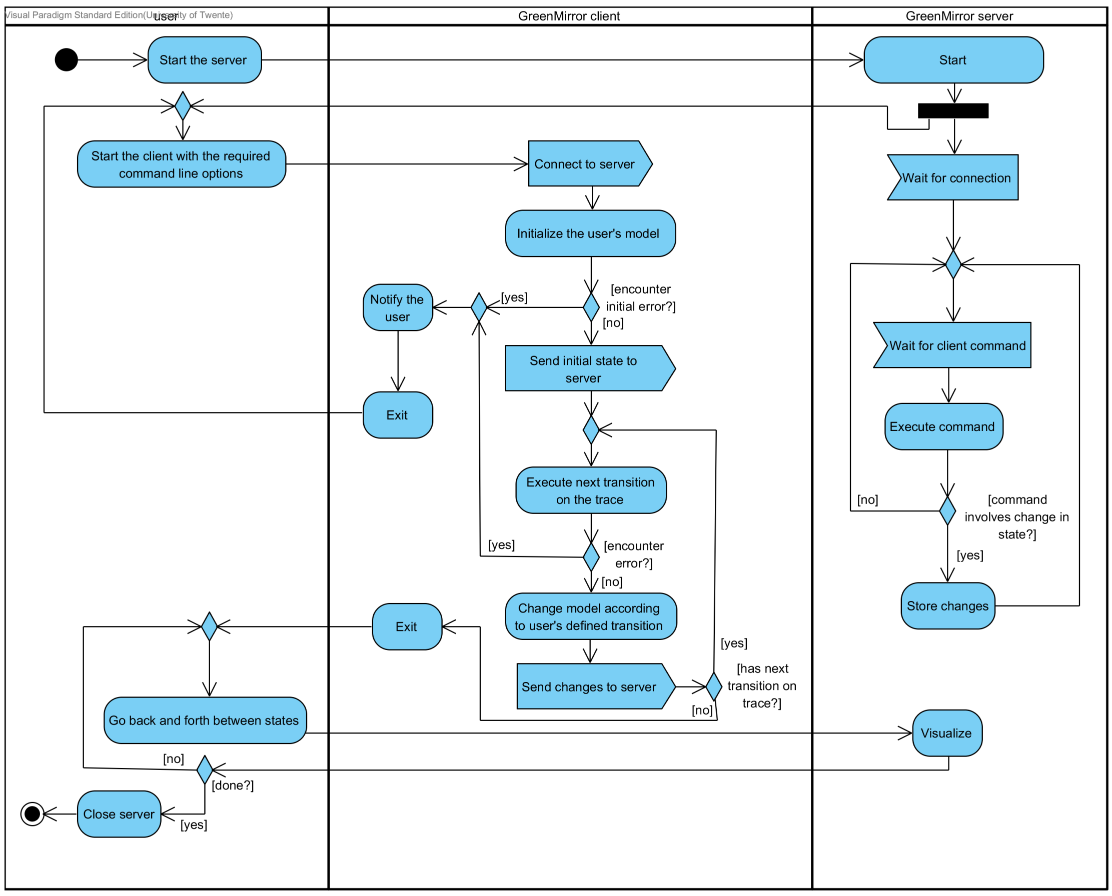

Visual Paradigm Standard Edition(University of Twente)
user
GreenMirror client
GreenMirror server
Start the server
Start
Start the client with the required command line options
Connect to server
Wait for connection
Initialize the user's model
[encounter initial error?]
[yes]
[no]
Notify the user
Send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

Figure 1: Simplified activity diagram of the general work-flow (see appendix A for a larger version)

3.3 Detailed work-flow

Let's start at the beginning: starting up the client or server and parsing the command line options. GreenMirror uses the JOpt Simple library (see [4]) 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 validating, parsing and processing of command line options, however, should not be understated. 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.

Now let's get into a little more detail about the specific 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 already have observed this and 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 2.

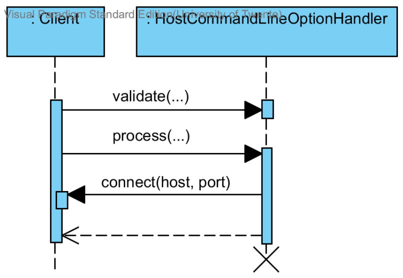


Figure 2: Simplified sequence diagram of the handling of the −−host command line option

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, every model initializer should, of course, 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 change 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 for more on this). Finally, when the model initializers have been executed and thus the initial state has been defined, GreenMirror sends the "end transition" command to the server, indicating that the transition from state 0 (no state) to state 1 can be performed.

Next in line is the handler for the −−trace option. 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 "end transition" 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 previous or 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 server sent. Every command is passed to the correct CommandHandler in the sequence as shown in figure 3. What exactly happens in the handle(CommunicationFormat, String) method of the CommandHandler entirely depends on the received command.

After the client has sent the "start visualization" 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 (figure 18 in appendix B) is that all visualization parameters of the state-transition are derived from the button the user clicked on. More on this will be explained in section 3.4.

3.4 Implementation details

Several implemented design choices and applied design patterns deserve more elaboration.

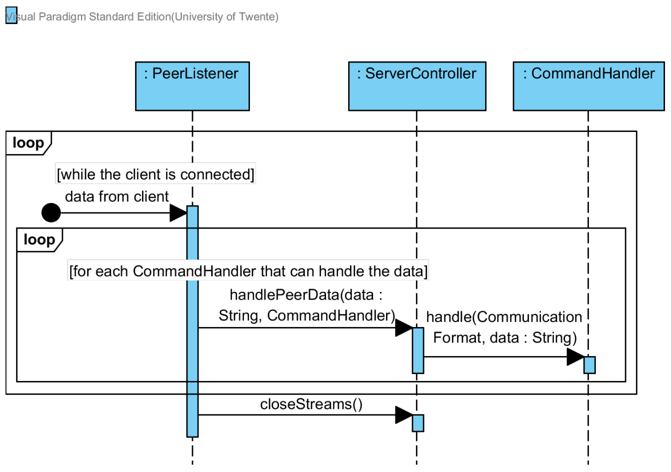

Visual Paradigm Standard Edition(University of Twente)
: PeerListener
: ServerController
: CommandHandler
loop
[while the client is connected]
data from client
loop
[for each CommandHandler that can handle the data]
handlePeerData(data : String, CommandHandler)
handle(Communication Format, data : String)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

Figure 3: Simplified sequence diagram of the server receiving data

3.4.1 State and state-transition tracking

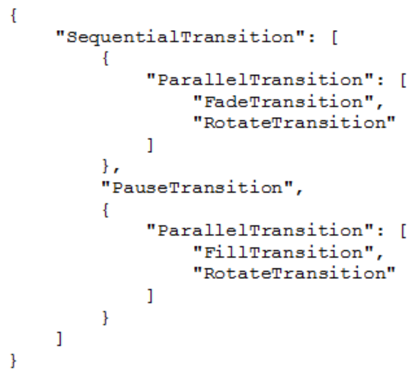
For future extensions, one of the most important parts to understand about the GreenMirror framework and application is the way it currently stores the state and state-transitions of the model internally. The client controller accepts changes in the model and immediately notifies the server of these changes. It also lets the server know when a state-transition ends, but it does not store the distinct states because this has no practical use in the current version (although it would be easy to implement by storing the state every time a ModelTransition finishes execution). On the other hand, the server does have a practical use for storing the distinct states: the user can browse through the states. This, in combination with the fact that the main goal of the GreenMirror application is the visualization of state-transition models, results in the need for the server to store and track both the distinct states and the transitions between the states. Because storing different internal states of an object is a well-known programming scenario, the *memento design pattern* (from [5, 8]) has been chosen to handle storing and restoring the visualizer states.

This pattern is implemented in the Visualizer and VisualizerMemento classes. Needless to say, VisualizerMemento fulfils the memento role, encapsulating the state's nodes, their relations and the state-transition data to go to the next state. It should be noted that the current version does *not* store the state's nodes and relations, because the way of storing state-transition data is sufficient. This can still be easily implemented in future versions if need be, due to the explicit use of the memento pattern: the Visualizer class fulfils both the caretaker and originator roles, implementing the VisualizerMemento.Caretaker and VisualizerMemento.Originator interfaces to make this more expressive.

State-transition data is stored as instances of subclasses of the javafx.animation.Transition class. Let's first expand some more on JavaFX' transition[2] classes. JavaFX makes animated transitions very simple to implement. When defining one's own, javafx.animation.Transition must be extended and only the interpolate(double) method must be implemented. The argument of type double is a fraction between zero and one, zero being 0% and 1 being 100%, indicating how far along the transition is. In this method, the value of the property that is animated should be set according to the fraction supplied by JavaFX and the starting and ending values of the property. There are several ready-made transitions available, such as FadeTransition and RotateTransition. JavaFX also has two special transitions: ParallelTransition and SequentialTransition. These play other transitions that are added to their lists in parallel or sequentially, respectively, and can be nested.

[2]note the distinction between JavaFX transitions and state-transitions. See section 1.2

This is also how GreenMirror stores a state-transition: one SequentialTransition holding one or multiple ParallelTransition s (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 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 entry 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 4.



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

Figure 4: An example of one stored state-transition in JSON notation

A reason to store only these Transitions in the memento is the fact that reversing the animation of a Transition is as easy as setting its rate property to a negative value. This makes it simple to go back and forth between states without the need to recalculate property values of JavaFX nodes.

3.4.2 FxWrapper and FxPropertyWrapper

Each GreenMirror node that has a visual appearance needs to store and track the properties of its FX. This is done using the *proxy design pattern* (see [5, 8]), with subclasses of the abstract FxWrapper class. These subclasses are also loaded using the ServiceLoader class. Using a wrapper in stead of directly using a JavaFX node instance has several 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 mentioned in section 3.4.1, 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 abstract FxShapeWrapper subclass adds support for JavaFX nodes that extend javafx.scene.shape.Shape, such as the fill property.

As just stated, it is easy to implement support for the use and visualization of a new type of JavaFX node to the framework:

1. Create a new subclass of FxWrapper or FxShapeWrapper.
2. Implement all necessary methods and override the relevant ones.

3. Add the JavaFX node properties you want to support (see below).
4. Add the binary class name of the new subclass to META−INF/services/greenmirror.FxWrapper.

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. This will be clarified with an example. 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 (more on this in section 3.4.3). 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? To make this modular and easily extendible, the abstract FxPropertyWrapper class handles this, also implementing the proxy design pattern. Adding support for properties to an FxWrapper now becomes more general and easy and results in these steps:

1. If no FxPropertyWrapper exists yet for the type of the property, create and implement it.
2. Add the property to getAnimatableProperties() if it can be animated or to getChangableProperties () if it can only be set once. These two methods are the overridden ones in the FxWrapper subclass the property belongs to.
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 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 (which will be explained in section 4), add an import to greenmirror.client.modelinitializers.GroovyScriptModelInitializer#IMPORTS.

From the above description and section 3.4.1, one might realize that this design has a notable constraint: JavaFX node properties that are changed after they are initially set *must* be animatable. However, the obstacle of animating the change of a discrete property can be overcome by implementing a Transition that animates a fade-out, changes the value, and animates a fade-in to the original opacity. This solution is used to animate changes in the image property of ImageFxWrapper and the text property in TextFxWrapper.

3.4.3 Communication between client and server

The abstract Command and CommandHandler classes respectively construct and interpret data exchanged between the client and the server. Every Command instance received the necessary data in its constructor, remembers it and optionally performs some preparation before the controller requests the data as a string, formatted according to the set CommunicationFormat. Currently, the JSON format is used. The controller prepends the name of the command before the data, so the peer on the receiving end knows which CommandHandler to use.

In the current version, only communication from the client to the server is required. The server works on the basis of its received commands. This means two things. In theory, the server could get completely different functionalities, based on new implementations of CommandHandlers. Secondly, the server must receive the initialization command before any other command that does something with the visualizer. In the current design this is the responsibility of the client, and specifically the model initializer implementations. New implementations of CommandHandler can be added by adding them to the ServiceLoader (by adding them to META−INF/services/greenmirror.CommandHandler) and making sure the implementation has at least one of the @ServerSide or @ClientSide annotations, indicating on which side the CommandHandler should be used.

3.4.4 Placement of nodes with respect to other nodes

As already mentioned, 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 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.

`NoPlacement` The default for a relation.

Every type of FxWrapper must calculate the exact coordinates of a specific placement. The FxWrapper class provides a static method to calculate these coordinates for a rectangle, given the width and height.

These placements are an important aspect of the visualization of nodes and their relations. They make sure the user doesn't have to work with actual coordinates in his model, except while defining the initial state, thus they provide a level of abstraction. Although the current implementations offer often used placements, a developer might want to add more distinct ones. This can be done by adding his implementation to META−INF/services/greenmirror.Placement so it can be loaded by ServiceLoader, and by adding the proper calculations to every FxWrapper type.

3.4.5 The visualizer's playback states

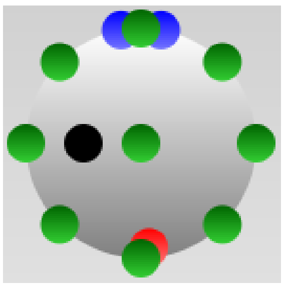


Figure 5: An example available placements on a circle FX node. The green circles have a corner* (or at least, what would have been the corner), edge* or middle placement, the blue ones an edge placement with -10 and 10 degrees, the black one a custom placement with relative position $(-10, 0)$ and the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

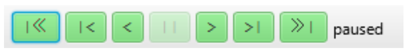


Figure 6: The toolbar when the visualizer is in the paused playback state and has previous and next model states

After the visualizer has been initialized, it can be doing one of three things: it can be waiting for the model to load, it can be waiting for the user to interact, or it can be playing an animation. The latter two will be discussed here. Note the explicit difference between the model's state and the visualizer's playback state.

From figure 18 (discussed in section 3.3) can be seen that the operation of the different toolbar buttons is determined directly after the playback state has been changed. This means that the behaviour of a button depends on the playback state (and the availability of a previous or next model state), resulting in the use of the *state design pattern* (see [5, 8]). Every playback state has its own class that implements the Visualizer.PlaybackState interface.

A reciprocal action can also be seen in figure 18: when the user clicks a button, the playback state changes. This changes the playback state as illustrated in the state machine diagram of figure 7. As one might notice, there are five states in figure 7, but seven buttons in figure 6. This is because the most left and most right buttons respectively rewind and forward through the model states. These result in a "stepping" playback state, with the difference that the JavaFX transition is played with a highly increased speed to make them nearly instantaneous.

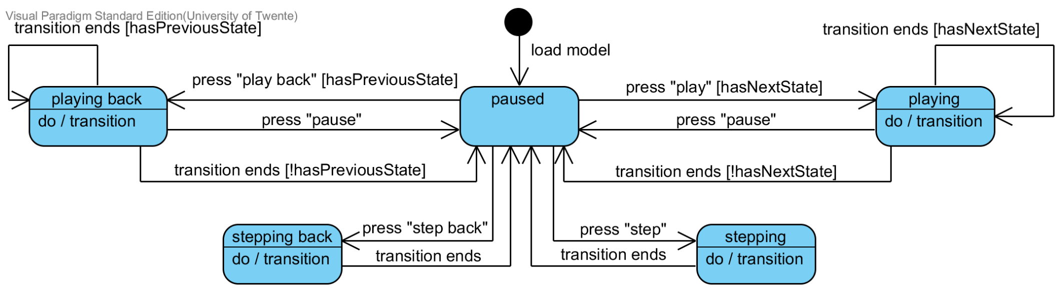
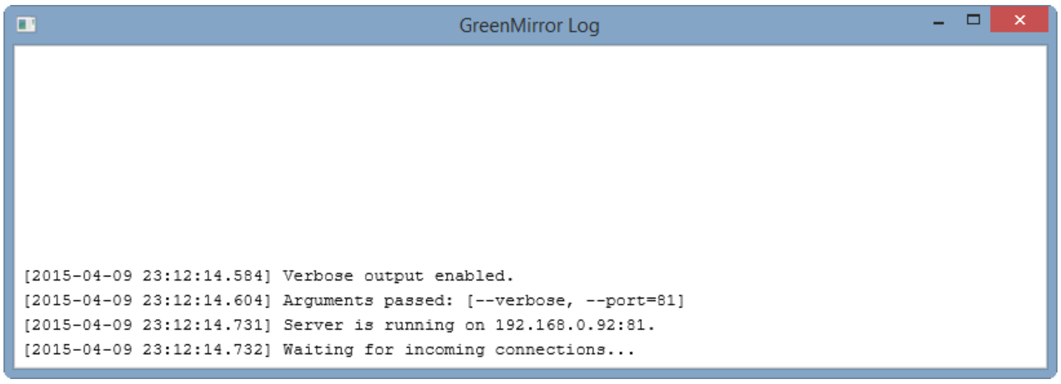

Visual Paradigm Standard Edition(University of Twente)
transition ends [hasPreviousState]
load model
transition ends [hasNextState]
press "play back" [hasPreviousState]
playing back
do / transition
paused
press "play" [hasNextState]
playing
do / transition
press "pause"
press "pause"
transition ends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

Figure 7: The state machine diagram for the playback states

3.4.6 Miscellaneous

A singleton Log class is available that accepts any implementation of PrintStream as its output by way of the *strategy design pattern* (see [5, 8]). Multiple can be used simultaneously and are selected at the entry point of the respective client and server. The client doesn't have a visual user interface, so it only selects the default System.out. The server is mainly visual, so it uses both the default System.out and the GreenMirror WindowLogger class to create a window that shows the log entries (see figure 8). Log entries are prefixed with a time stamp, including the milliseconds so the timing can also be analysed. The log also has a verbose setting: when set to true, a lot more information is provided about what exactly is happening internally.



```
[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 8: An instance of the WindowLogger

When a state-transition removes a JavaFX node from the visualizer, it doesn't actually gets removed internally. It would be a hassle to remove and re-create the JavaFX node each time the user respectively reaches the states and comes back to it. This would also cause several other difficulties, such as the fact that GreenMirror needs to know that a JavaFX node has to be added *before* it transitions back to the state (so it can animate the creation). That's why JavaFX nodes are not actually removed: they are merely made invisible by changing the opacity to zero. However, in anticipation of future extensions, the JavaFX nodes are also made untouchable for mouse events. This way, invisible nodes don't become an invisible layer preventing mouse events on other nodes. Before the start of each JavaFX transition, GreenMirror checks if the JavaFX transition tree contains a FadeTransition that fades a JavaFX node in from an opacity value of zero. If so, it sets the visibility property to true. After a JavaFX transition has ended, it checks if the JavaFX transition tree contains a FadeTransition that faded a JavaFX node to opacity value zero. If so, it sets the visibility property to false, making it practically not there as far as the user knows. When a GreenMirror node is removed in the model of the client, it is replaced by an instance of NullNode, following the *null object design pattern* (see [5, 8]). This makes sure the model throws expected exceptions and ensures the user will get properly notified if he tries to access it.

GreenMirror has an auxiliary GridBuilder class created to take away the tedious work of building a grid of nodes. As one might guess, it follows the *builder design pattern* (see [5, 8]). 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 a GreenMirror node per cell and one extra for the background, all of which have RectangleFxWrappers. Listing 1 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. The result result is visible in figure 9.

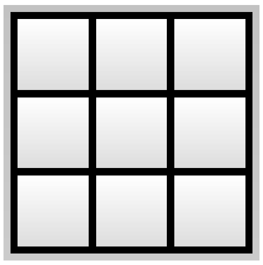


Figure 9: The result of the GridBuilder example code of listing 1

Listing 1: Example code 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()
```

Although both the client and the server work with multiple threads, there is no need for extensive measures to ensure thread safety. As the client currently does not receive data from the server, everything is executed on the main thread. The server, on the other hand, does work with a JavaFX thread and a PeerListener thread. However, seeing as the JavaFX thread starts visualizing after the complete model has been received from the client, and the user interaction with the visualizer is limited during visualizations (as is explained in section 3.4.5), no concurrency can occur.

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, and the usage of the implementations follows the *strategy design pattern* (see [5, 8]).

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 this information is up to the implementation. Once it has received this information, it can add nodes to the Client controller, remove nodes, 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. It is not recommended that the model initializer uses this to directly convey model changes, 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 SetAnimationDurationCommand, which sets the duration of all animations created as a result of upcoming model changes.

The model initializes should define what model changes should happen by adding a new instance of ModelTransition to the controller. As mentioned in section 3.3, this class has a groovy.lang.Closure field that changes the model when executed. This is type is chosen to directly support the first implemented model initializer (discussed below), although it is not restricted to the first implementation. The closure can accept arguments based on the trace transition. 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(\d+)$ 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.

As mentioned in section 3.4.3, the server could be completely re-purposed by implementing and using different Commands and CommandHandlers. However, if the server will be 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. Of course, the implemented model initializer could delegate this responsibility to the user.

The model initializer that has been implemented in this first version of GreenMirror is based on Groovy (see [1]). 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 selects a different 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 2 shows an example of a user script that can be executed by the GroovyScriptModelInitializer. Figure 10 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 2: Example Groovy user script

```
initialize(600, 400);
addNodes(
    new Node("loc:1").set(fx("rectangle")
                        .setSize(200, 400).setPosition(0, 0)
                        .setFill("radial-gradient(center 100px 200px, "
                                  + "radius 200px, white, red)")),
    new Node("loc:2").set(fx("rectangle")
                        .setSize(200, 400).setPosition(400, 0)
                        .setFill("radial-gradient(center 500px 200px, "
                                  + "radius 200px, 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:"))
    );
});
```

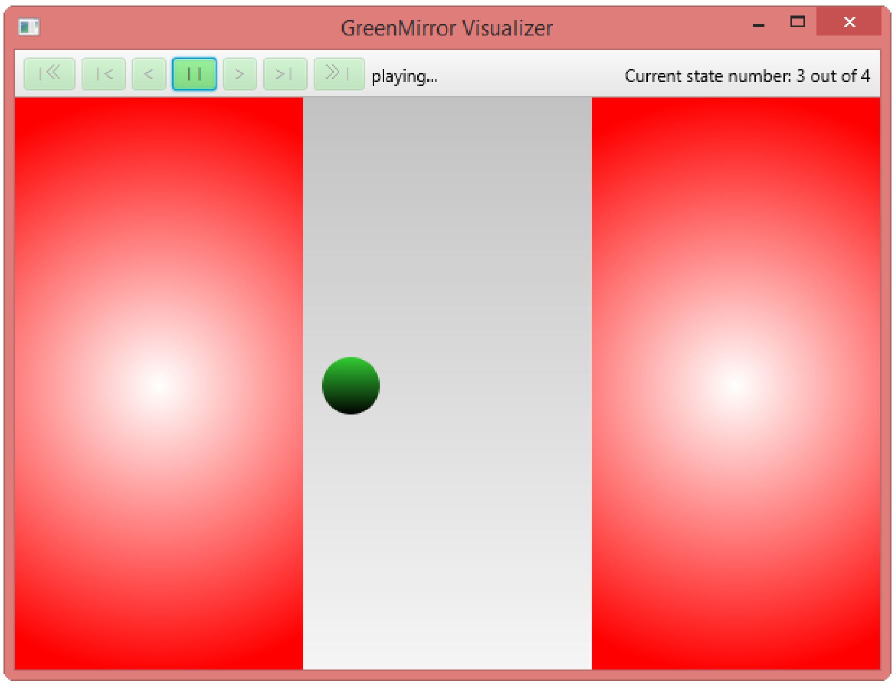


Figure 10: A screenshot of the executing visualization of listing 2 with trace switch, switch, switch

After the model initializer has set the initial model state and saved all Model−transitions, the selected TraceSelector gets executed. Section 3.3 already elaborated on the sequence of events in which a TraceSelector is executed. The current FileTraceSelector implementation retrieves the trace from a simple file where the transitions are separated by a newline. In the visualization example of figure 10, the trace file looked like listing 3. In the toolbar of figure 10 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 3: Example trace file

```
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 trace, respectively. In GroovyScriptModelInitializer this is the file name of the Groovy script, while in FileTraceSelector this 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.

5 Validation

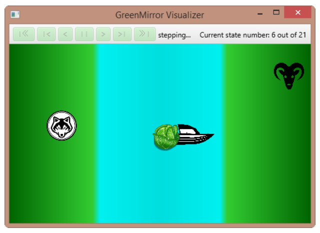


Figure 11: A screenshot of the visualized ferryman puzzle

Due to the visual nature of the application, most tests are based on visual inspection. Several unit tests have been written using the JUnit library. These test the essential model classes used by the client and server: Node, Relation, NodeList and RelationList. Furthermore, the FxWrapper class and its currently implemented subclasses have been unit-tested to validate their correct functioning with the client. In addition to these subclasses, a separate unit test has been written to validate the calculations used for the placements per FxWrapper type. The proper functioning of these classes is essential for the user to be able to create a model that can be researched with GreenMirror.

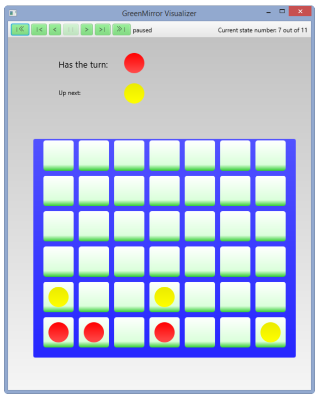


Figure 12: A screenshot of the visualized ConnectFour game

Other validation tests have been done with the GroovyScriptModelInitializer using comprehensive Groovy scripts. These scripts enable the user to validate the workings of GreenMirror by inspecting the visual results. The Groovy scripts testcases/systemtest1.java and testcases/systemtest2.java located on the repository of this project, together with their respective trace files found in the same directory, extensively validate the possibilities of GreenMirror. They include tests from simply adding nodes and relations in all possible ways, to moving multiple chained, rigidly placed nodes around.

A few test models were described in section 2.5. These were also used as system tests and are also available in the testcases directory on the repository. Screenshots of the resulting visualizations can be found in figures 11 to 13 for the ferryman puzzle, the ConnectFour game and the Dining Philosophers problem, respectively.

6 Discussion

The GreenMirror application is the first step in the creation of an extensive research tool. This section will discuss several ideas for future improvements. First and foremost, it should be stated that future versions should be designed or altered into a more modular and structured way and by a more experienced developer. Although the current version does fulfil its intended purpose, the framework hasn't even begun to reach its full potential.

The log functionalities could be extended and modified to present event data in a more structured way. Several event 'levels' could be added, such as regular messages, warnings and fatal errors. To use the log as a proper debug tool for the researched model, snapshots of the system state could be included, possibly in an additional output sink.

To make the client more user friendly to work with, a graphical interface could be added. This is in no way necessary with the current functionality. It would just provide more convenience for the user and would be more appealing for the eye.

A caching function on the server might result in a slight performance improvement when handling models consisting of over hundreds of state-transitions. The performance improvement

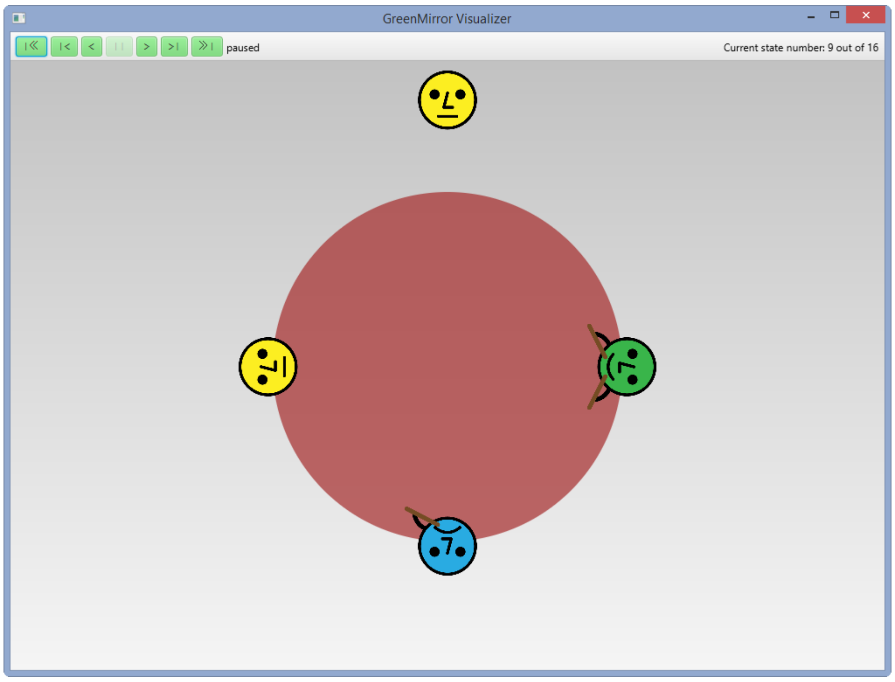


Figure 13: A screenshot of the visualized Dining Philosophers problem

would reside in the fact that the trace selector wouldn't have to process all state-transitions every time the model is loaded, but it would retrieve the already-constructed animations and states from a cache file. This also means that every time the client knows it's going to process a model, it sends a checksum of the model and the trace to the server, verifying if processing is necessary at all. The user would in practice alter his model or trace a lot, which raises the question how often this caching function would be used.

JavaFX has extensive 3D visualization possibilities. These could provide more eye-appealing visualizations for the simple models presented in this report, but could also be an essential asset in visualizing more complex models. The current version of GreenMirror already supports the extra z-coordinate by working with javafx.geometry.Point3D objects in stead of javafx.geometry.Point2D objects to represent positions on the visualizer . A developer could fairly easily extend the framework so it can work with 3D shapes. Together with some other small modifications, this can be accomplished implementing new FxWrapper and Placement subclasses.

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 Groovy 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. Should such an implementation be developed, the user might be required to provide extra information about how GROOVE's nodes should be represented on the visualizer. Fortunately this can be done rather simple: the user can provide a script that uses the Groovy ModelInitializer implementation to supplement the model, which is possible because multiple model initializers can be used in GreenMirror.

Continuing this line of thought: why not write a supplement to the Groovy model initializer that constructs a GROOVE model (or model for another tool) from the model the user defines in his script? This would not be easy due to the different ways the tools work, but shouldn't be impossible. This way, a model could be analysed by writing it just one time in stead of writing it two times for two different tools.

To conclude this section some recommendations must be given to the developers and researchers that might want to further develop the GreenMirror framework. First, future developers should take a critical look at the degree of structure implemented in the design of the framework and specifically whether it is sufficient for future development. Secondly, as is discussed in section 2.4, it is a very large project to develop extensive support for generating a model based on interaction with the visualizer. Therefore, the suggestion is to first, or perhaps rather in parallel, develop implementations of ModelInitializer and TraceSelector to connect other tools to GreenMirror.

References

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

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

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

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

[5] Partha Kuchana. *Software architecture design patterns in Java.* CRC Press, 2004.

[6] Jeff Magee and Jeff Kramer. *Concurrency: state models & java programs.* John Wiley & Sons, 1999.

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

[8] Alexander Shvets. *Design Patterns Explained Simply.* Sourcemaking.com.

A Activity diagrams

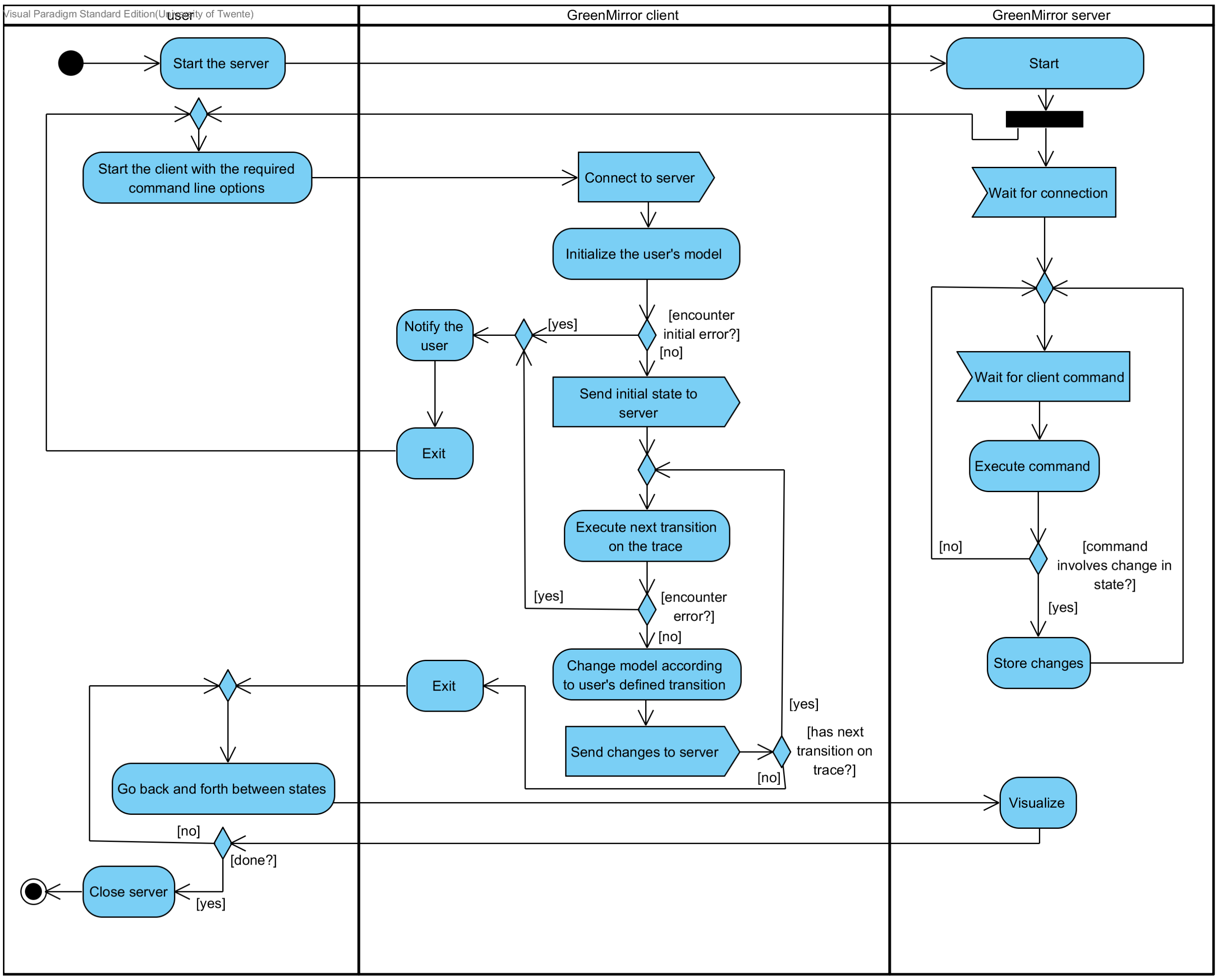
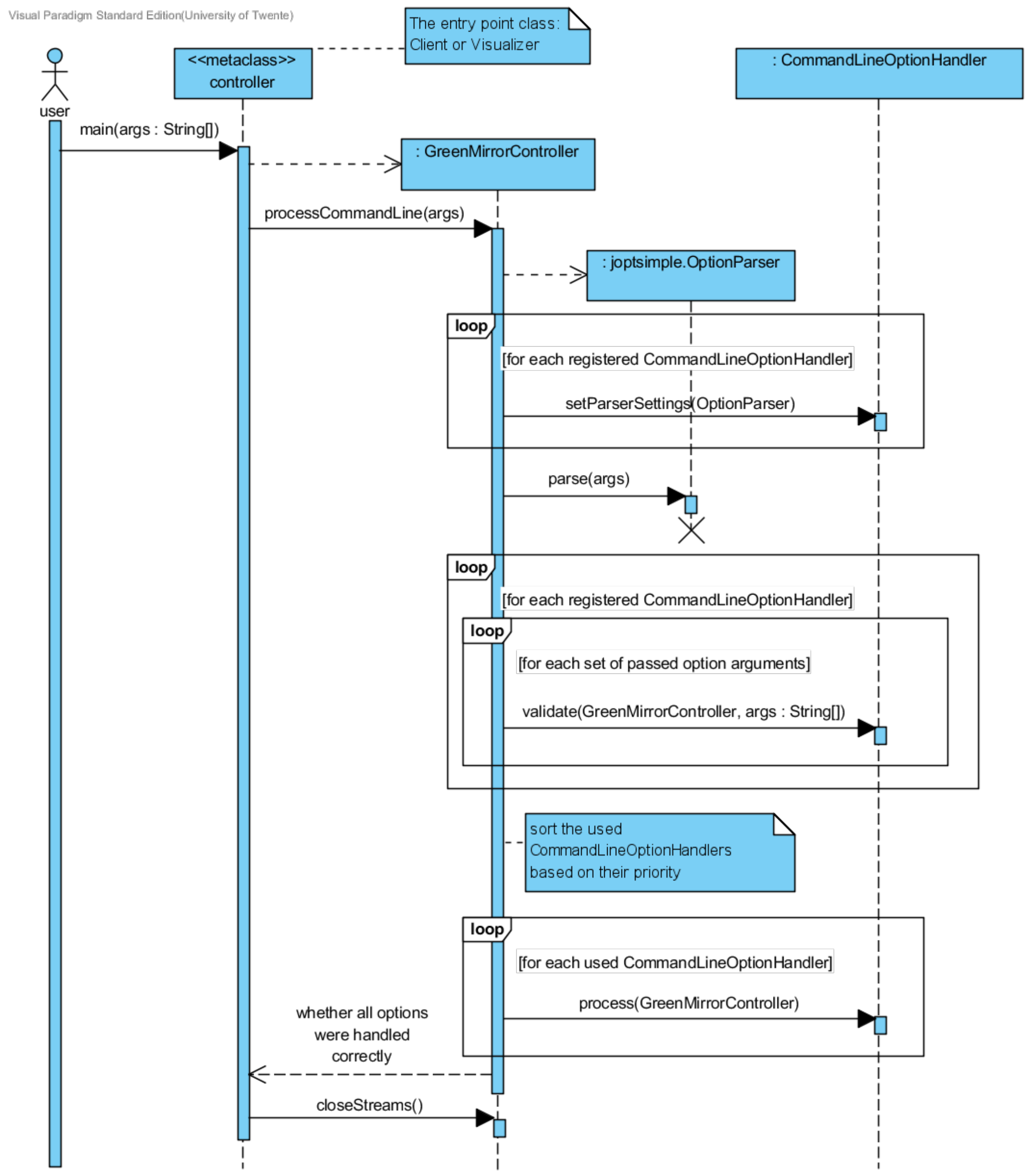

Visual Paradigm Standard Edition(University of Twente)
user
GreenMirror client
GreenMirror server
Start the server
Start
Start the client with the required command line options
Connect to server
Wait for connection
Initialize the user's model
[encounter initial error?]
[yes]
[no]
Notify the user
Send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

Figure 14: Simplified activity diagram of the general work-flow

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
user
<<metaclass>> controller
: CommandLineOptionHandler
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.

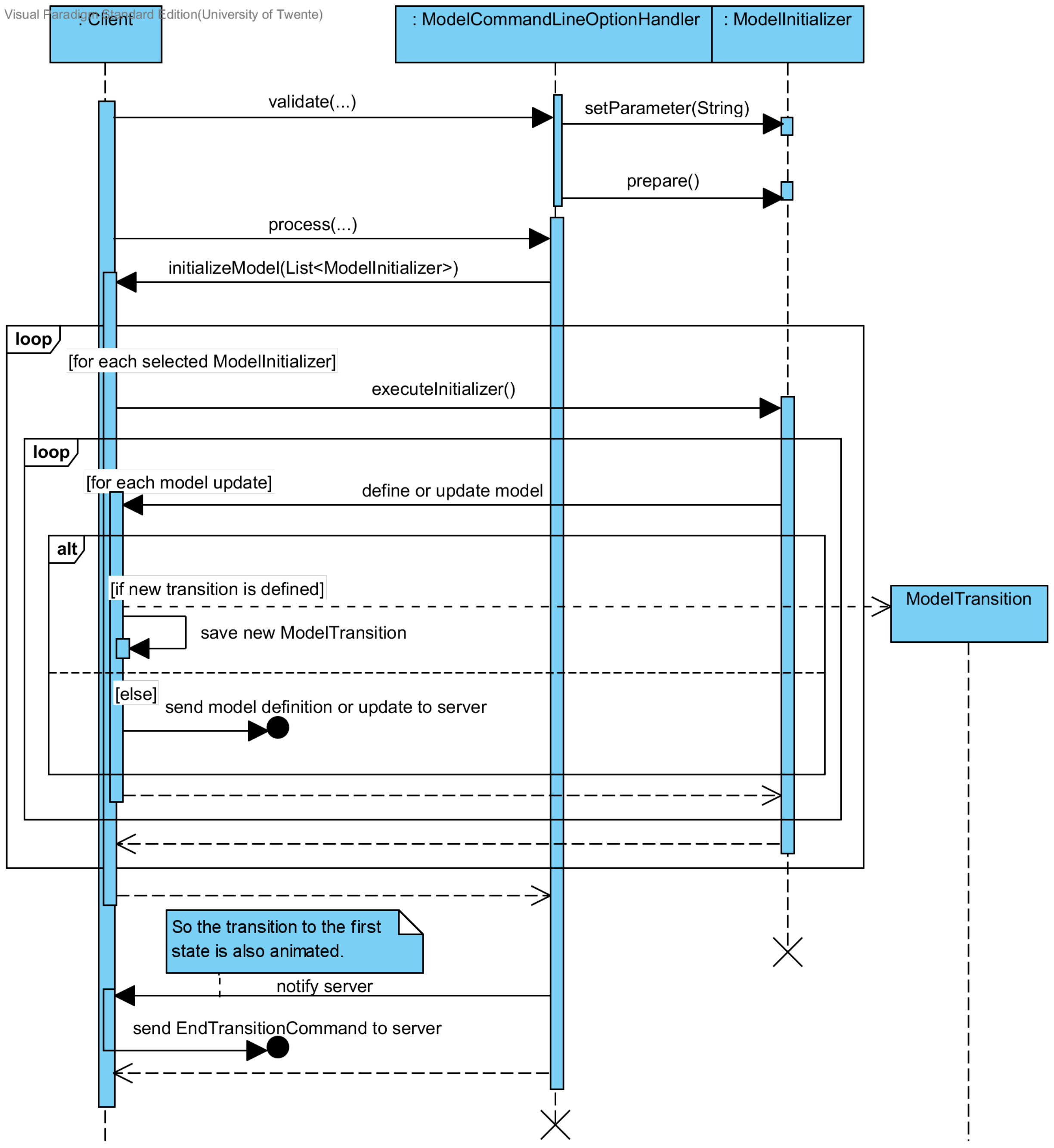

Visual Paradigm Standard Edition(University of Twente)
: Client
: ModelCommandLineOptionHandler
: ModelInitializer
validate(...)
setParameter(String)
prepare()
process(...)
initializeModel(List<ModelInitializer>)
loop
[for each selected ModelInitializer]
executeInitializer()
loop
[for each model update]
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

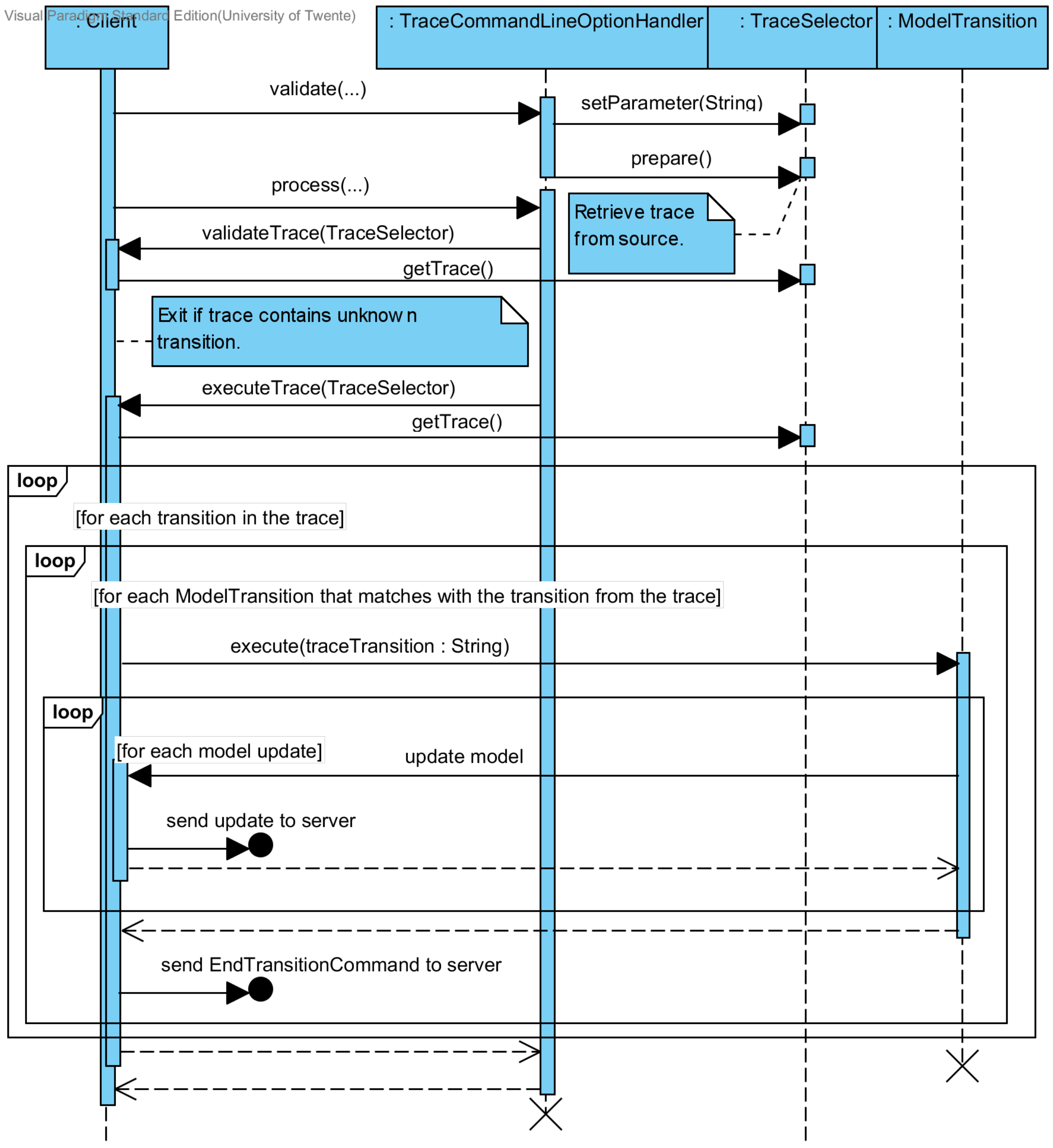

Visual Paradigm Standard Edition(University of Twente)
: Client
: TraceCommandLineOptionHandler
: TraceSelector
: ModelTransition
validate(...)
setParameter(String)
prepare()
process(...)
Retrieve trace from source.
validateTrace(TraceSelector)
getTrace()
Exit if trace contains unknown transition.
executeTrace(TraceSelector)
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

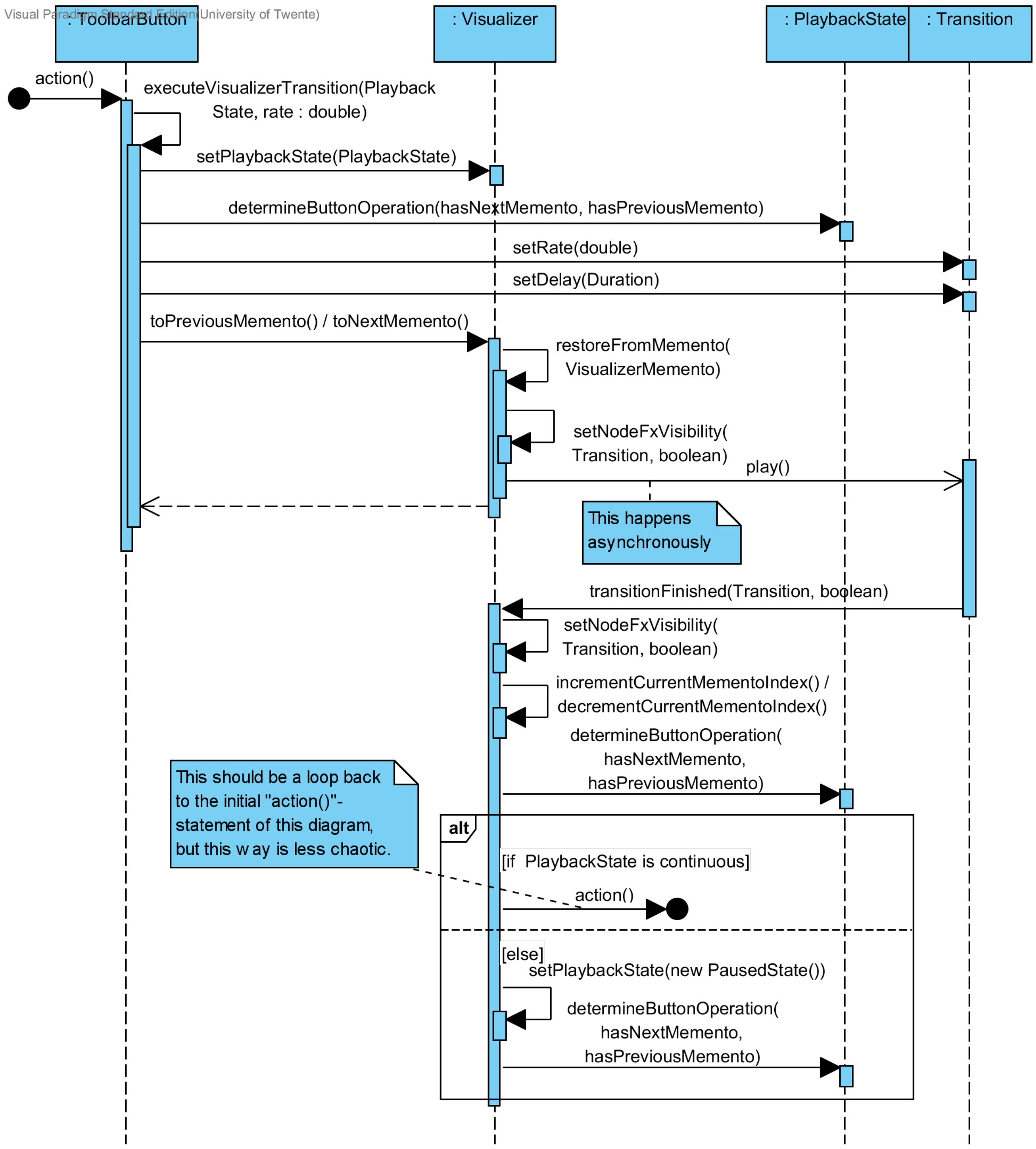

Visual Paradigm Standard Edition(University of Twente)
: ToolbarButton
: Visualizer
: PlaybackState
: Transition
action()
executeVisualizerTransition(Playback State, rate : double)
setPlaybackState(PlaybackState)
determineButtonOperation(hasNextMemento, hasPreviousMemento)
setRate(double)
setDelay(Duration)
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)