Design: Animator

DaCoPAn

Helsinki 31st May 2004

Software Engineering Project

UNIVERSITY OF HELSINKI
Department of Computer Science

UNIVERSITY OF PETROZAVODSK
Department of Computer Science

Course

581260 Software Engineering Project (6 cr)

Project Group

Carlos Arrastia Aparicio
Jari Aarniala
Alejandro Fernandez Rey
Vesa Vainio
Jarkko Laine
Jonathan Brown

Kirill Kulakov
Andrey Salo
Andrey Ananin
Mikhail Kryshen
Viktor Surikov

Customer

Markku Kojo

Project Masters

Juha Taina (Supervisor)
Yury Bogoyavlenskiy (Supervisor)

Turjo Tuohiniemi (Instructor)
Dmitry Korzun (Instructor)

Homepage

`http://www.cs.helsinki.fi/group/dacopan`

Change Log

Version	Date	Modifications
1.0	April 15th 2004	First version, will be updated along the implementation phase.

Contents

1 Introduction

The DaCoPAn Animator module enables the visualization of the packet trace information provided by the DaCoPAn Analyzer module through the Protocol Events File (PEF). This Animator Design Document is meant to be a guideline for the implementation of the Animator module.

Section 2 presents the general architecture of the Animator module. An overview diagram will introduce us to the different elements of the Animator, explaining how they are interrelated. A description of the elements can be found after the diagram. However, more precision on the different components follows in the next sections.

Section 3 contains a description of the different data structures that will hold the packet trace information given by the Analyzer module. These data will be made accesible to the other components in a suitable way. Data related to the scenario can also be included in the data structures (e.g. notes).

Section 4 introduces the main user interface design. The different areas that the DaCoPAn Animator main window will show, their different features and the toolbar needed to control them are presented. Java Swing package will be used for implementation (javax.swing).

Section 5 contains an explanation about the Animation setting classes.

Section 6 explains how the Control signal framework is in charge of receiving commands from the user and timing events, and calling consequently the appropriate DaCoPAn Animator components, through its different classes and interfaces.

Section 7 contains the necessary information about the Animation Sequence framework, that is how DaCoPAn Animator manages to present different presentations sequentially.

Section 8 briefly presents how notes to be shown during different animation types are handled by the Animator module. Notes are not a separate module, but may be included in some other Animator component.

Section 9 presents the 3 different animation panels that the Animator will show, Message Sequence Chart, Unit flow orchestration and Encapsulation. It contains both diagrams and text descriptions on the disposition and behaviour of these.

Section 10, Protocol events file reader, illustrates how the Animator will parse the packet trace information present in the PEF (Protocol Event File) using existing XML libraries. Still, the PEF reader won't be tied to XML format for further developments of the DaCoPAn software.

Section 11 presents a way to save and load scenario data to and from a file, by representing different settings and configurations made by the user and related to a specific set of packet trace data (to a PEF, basically).

Section 12 describes the localization architecture of the Animator.

2 Architecture

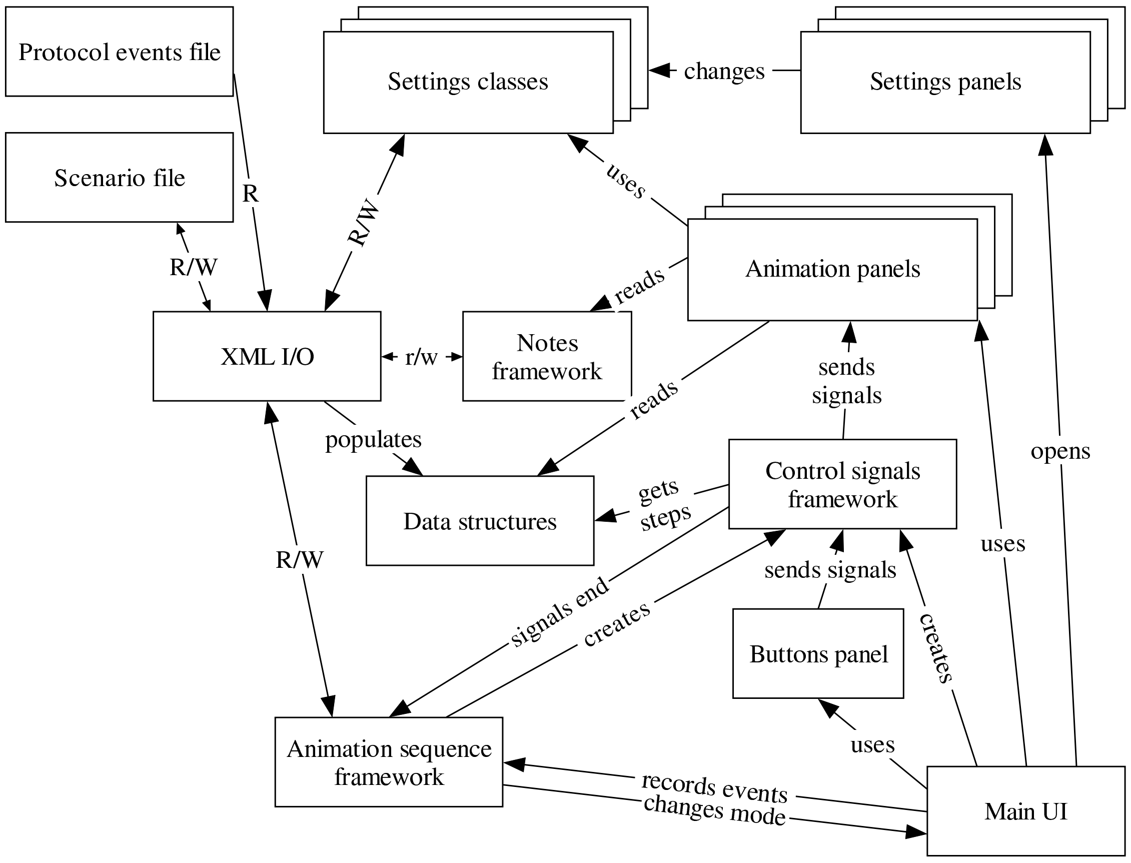


Figure 1: Architecture diagram

2.1 Data structure classes

Store all animation data internally and provide a convenient view to this data for all other components. After the data has been read from the protocol events file, the data does not change, so there will be no need for update or delete operations, or concurrency problems, at least for network data.

There may also be a small amount of data specific to the animation, e.g. some kind of notes. This data may need to be mapped to the network data and be accessible also through the view to the data.

2.2 Main UI

Means first of all the classes for the main application frame. The classes are responsible for the layout of the animation components and making most commands available to the user, e.g. through a menu bar.

This set of classes need to provide methods for e.g. changing the view mode (to Enc, to MSC).

The Main UI will also need to be able to provide signals about user actions (mode change, start, stop, etc.) that are needed for recording animation sequence information.

2.3 Animation panels

Individual panels (derived from javax.swing.JPanel) for presenting different types of animation. Each panel acts on signals from control signals framework. Each panel queries the necessary data from the data structures. Thus, these panels are independent of each other, and rather passive in terms of user interaction. The individual panels will be MSC, Enc and possibly UFO and TPI.

2.4 Settings classes for animation

Objects will be mostly record-like collections of relevant settings for a particular animation type (or mode, e.g. MSC+UFO). Classes will possibly need to

provide methods for validating user input. The objects should also be clonable to facilitate storing of an animation sequence.

2.5 Panels for changing settings

A set of JPanels that can be used by the user to change settings in the settings objects for animation panels. There should probably be a superclass to provide form-like functionality and individual classes for each animation type and possibly other settings.

2.6 Control signals framework

Provides a modular and uniform way to map control signals from user (selecting buttons like Play, Pause, To Beginning, To End) and from an animation timer to the animation panels. One or many animation panels may be simultaneously controlled by the framework.

2.7 Buttons panel

An instance of javax.swing.JToolBar, that contains the player control buttons for (any) animation, and sends signals (method calls) about user actions to the control signals framework.

2.8 Animation sequence framework

Is able to record and store a sequence of different presentation types (e.g. MSC

animation, Encapsulation animation). Recording is done by collecting relevant user actions from the main UI and storing the necessary settings objects. There will need to be functionality to enable the control signals framework to signal the end of one presentation type so that the animation sequence framework can make actions to start the next presentation type. The format for storing the sequence

is a playlist-type list of different presentations, that can also be presented to the user as a "playlist". There can be functionality to allow the user to jump to any item in the playlist and thus allow using the list as a crude "table

of contents" for a scenario.

The framework will also need to allow the user to include breakpoints (containing textual comments) to the animation sequence.

2.9 Notes framework

There may possibly be a separate framework to allow the user to input and edit notes to the animation. (This was previously considered part of animation sequence, but it doesn't necessarily have to be part of it.)

2.10 XML input/output

Takes care of reading in the Protocol events file from the Analyzer and populating the data structures with network data. Will also need to be able to write and read XML data from the Animation sequence framework and the Notes framework.

2.11 Localization

Means a way storing all localisable resources (mostly strings) in one place where they can all be edited in one place. The mechanism should allow for retrieving localisable resources according to the locale in use. The mechanism should possibly also allow for placing placeholders (for variables) inside strings. The basic mechanism for this is already provided by a standard Java class, java.util.ResourceBundle, so we only need to develop an intermediate class for accessing the localized texts.

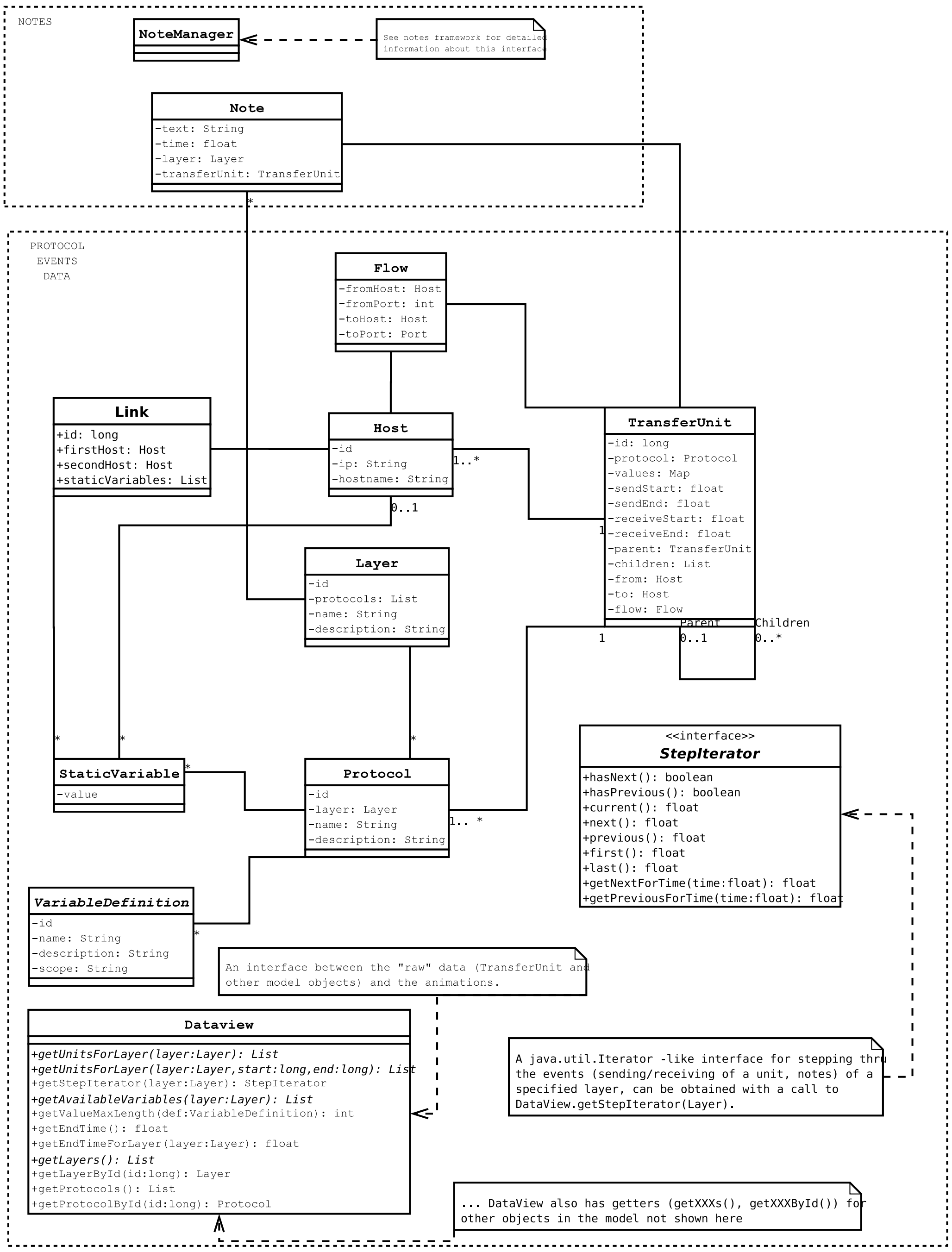


Figure 2: Class diagram for the data structures

3 Data structures

3.1 Protocol data

The central data structure class for the protocol data is called **TransferUnit**. A transfer unit is a generalization of a unit transferred on any network layer: for example, an IP packet, a TCP segment, or a HTTP request. An instance of this class encapsulates all the essential data on a unit: source and destination hosts, timestamps (send/receive), variable data (protocol-specific header fields, host variables) etc. The units on different layers form a tree structure (parent-child relationship) that's used to represent the encapsulation that occurs between different layers. For example, if the data of an HTTP response is contained in three separate TCP segments, the TransferUnit representing the HTTP response will have three children (the TCP segments) and each of these children will have the HTTP response as their parent (see Figure 3).

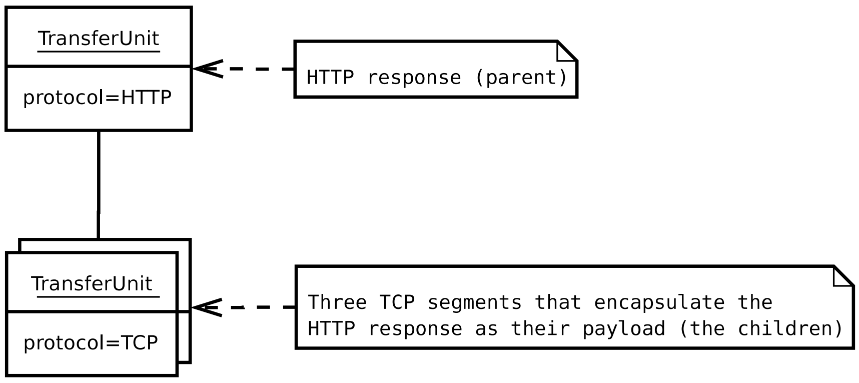

TransferUnit
protocol=HTTP
HTTP response (parent)
TransferUnit
protocol=TCP
Three TCP segments that encapsulate the HTTP response as their payload (the children)

Figure 3: Encapsulation in TransferUnit objects

In general, the data structures used to represent the protocol data in the Animator are immutable, i.e. they will not (or can't) be modified after they have been created. This in turn implies that they are thread-safe, and can be concurrently accessed by multiple threads without using synchronized collection classes, for example.

3.2 Host class

The **Host** class holds information about a single host, containing the hostname (for instance, "A") and the IP number in String format. A host is the network entity which interchanges packets (transfer units) with other hosts. References to hosts are maintained from the Transfer Unit and the Connection classes.

3.3 Flow class

The **Flow** class contains information about a protocol-specific connection between 2 hosts, e.g. a TCP connection. Therefore an instance of it makes references to both source and destination hosts, and as well stores information about the port numbers used for this connection to be established.

3.4 Link class

The **Link** class models a physical link between two hosts. It can have constants such as MTU (maximum transfer unit size) associated with it.

3.5 Protocol class

This class is intended to maintain data about the name and the network layer of a given protocol. One **Protocol** can only belong to 1 Layer, there must be at least one Transfer Unit for that Protocol and it can have none or many StaticVariables and VariableDefinitions

3.6 Layer class

The **Layer** class represents a network stack layer. It is described by the name commonly given to the layer. Many Protocols can belong to a same layer.

3.7 VariableDefinition class

This class contains all useful information needed to describe a network variable, meaning that it contains a variable's id (for animator and analyzer internal representation), the name of the variable, a short description and the variable scope. Depending on the scope of a variable, this one will be shown in different places during visualization. The name and description attributes could be determined by the Localization feature to adapt them to particular languages. A Protocol can have many or no variables. The **VariableDefinition** class might be used for those variables that can change throughout the interchange of Transfer Units between the hosts, for instance representing the value of the congestion window. Those dynamic variables must be mapped to Transfer Units as their value changes in relation to the packet interchange. For static variables that are constant all through the packet interchange sequence, the value is represented with a StaticVariable instance.

3.8 StaticVariable class

A **StaticVariable** is a particular kind of VariableDefinition that is specific to a Host and a Protocol. This class contains a value attribute that will store the actual value of the network static variable. It is called static as it doesn't change during the packet interchange. It extends from VariableDefinition.

3.9 Notes

Notes are separately discussed in section 8 Notes framework.

3.10 Managing the data

While a TransferUnit object contains all the necessary information that needs to be visualized of a single protocol unit, different animation types need a way of accessing the protocol data as a whole. The TransferUnit class has a way of representing the encapsulation between units on different layers, but has no mechanism for returning all the units on a specific layer in a sequential list, for example. This mechanism is needed by the MSC animation type, for example, as it is only interested in units on one specific layer at a time. Thus, there is a need for an interface that can provide different views to the "raw" unit data (TransferUnit objects) for the rest of the system.

This interface is called **DataView**. While the actual TransferUnit objects are created by the XML I/O classes when reading in the protocol events file, the DataView interface processes the units further to provide meaningful and efficient ways of viewing the data. For example, it indexes the units by their network layers so that retrieving an ordered list of units on the transport layer is a constant-time operation (see Figure 4). Furthermore, the DataView interface has methods that provide a way of accessing only a subset of data on a certain layer by allowing a time interval to be specified. For example, an MSC animation is not interested in drawing any units that would not be visible on the user's screen at a given moment, so it can ask the DataView to return only the visible units (the visibility of a unit is determined using its timestamps by the DataView).

3.11 StepIterator interface

For stepping thru the animations (MSC in particular) in a meaningful way, certain steps need to be calculated from the protocol data. A special interface, `StepIterator`, provides the animations with a means of stepping thru the events of a certain layer: sending and receiving units as well as any notes on the layer.

4 Main user interface design

4.1 General User Interface design

The main window of the DaCoPAn animator is divided in the following parts:

- **tool bar** for letting the user controll the animation
- **menu bar**
- **main animation area**
- **secondary area** which is divided between a Unit Flow Orchestration animation panel and a note area.
- **status bar** for showing information about the state of the animator

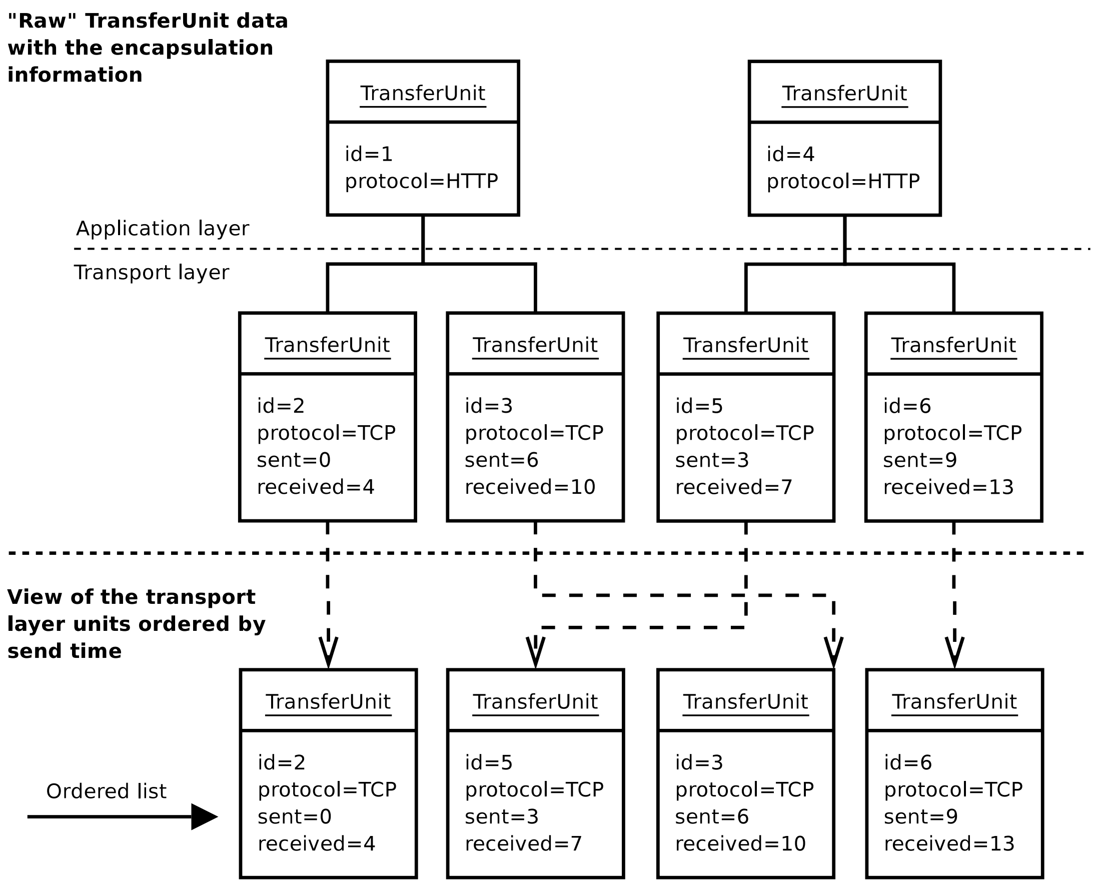


Figure 4: An example view on protocol data provided by the DataView interface

Most of the animation occurs in the main animation area. This animation is a sequence of Message Sequence Chart (MSC) type animation that is occasionally interrupted with some Encapsulation (ENC) animation. The animation can be watched in two modes: scenario mode and explore mode. In scenario mode the animation follows a predefined script and in explore mode the user is in charge of controlling the animation flow.

The main animation area takes up most of the screen width and is positioned on the right side of the program frame. The right side is used by the secondary area and divided vertically in two parts. The areas are divided using Swing Splitpanes so that the user can easily drag them to his preferred sizes. The animation panels then know how to draw themselves according to the space given to them.

When starting the animator, no animation panels are shown as there is no file loaded to the memory. Instead a welcome screen is shown in the main animation area.

4.2 The functionality of the main UI classes

The DaCoPAn animator is based on the Swing framework, which means that the control over the software runs always through Swing, which in turn gives the control over the other parts of the program to MainFrame, i.e. the main UI classes. When the animator is started an instance of MainFrame is created. It then creates the user interface using the class UserInterface and instances of all other classes needed by the different animation modes: DataView for accessing the data structures, NoteManager for using notes, AnimationSequence for storing scenario information. An instance of AnimationTimeState is created when the animation is started.

After this the animator is in a state called DACOPAN_STATE_NO_FILE and in a mode DACOPAN_NO_MODE which indicates that the animator doesn't do anything before the user uses the menu or tool bar to open a file.

MainFrame can be in the following modes:

- DACOPAN_NO_MODE: Used when no animation file has been loaded
- DACOPAN_EXPLORE_MODE
- DACOPAN_SCENARIO_MODE

MainFrame also has three distinct states:

- DACOPAN_STATE_NO_FILE: Used when no animation has been loaded
- DACOPAN_STATE_PLAYING_ENC: Used when the animator is showing Encapsulation animation
- DACOPAN_STATE_PLAYING_MSC: Used when the animator is showing MSC animation

Distinction between play and pause mode is done by AnimationTimeState and therefore those are not distinct states from the point of view of MainFrame.

To show the relations between MainFrame, other user interface classes and the rest of the animator, a class diagram is presented in Figure 5.

4.3 User actions

The menu bar contains the following commands for letting the user control the animator. Most of them are also present in a tool bar on the top of the main frame.

Open file This command opens a dialog for selecting a Protocol Events File or a Scenario file to be loaded to the animator.

Save file Clicking on this button saves the file in memory along with any note and scenario data included. If the file has not been previously saved as a scenario file, the user is prompted for the file to save.

Settings This command opens a settings dialog that is described in more depth in section 6 (Animation settings).

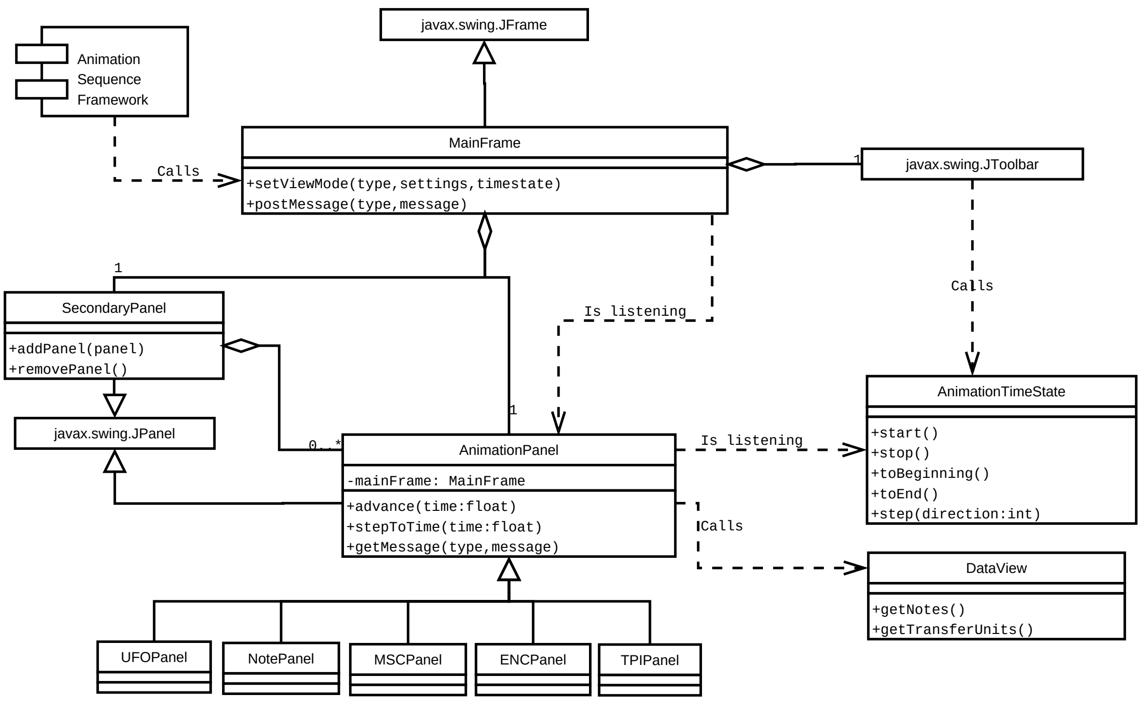


Figure 5: Main UI Class Diagram

Rewind to the beginning This buttons rewinds the animation to the beginning of the current AnimationTimeState. This command can only be called when the animation is paused.

Play Calls the current AnimationTimeState to start playing the animation it is assosiated with. If the current sequence is part of a bigger scenario, the AnimationTimeState notifies the animation sequence about reaching the end of the animation.

Pause Pressing this button changes the animation to paused mode.

Fast forward to the end This buttons steps to the end of the animation data assosiated with the current AnimationTimeState.

Step forward When in stop/scroll mode, this button can be used to step forward through events one by one in the animation.

Step backward When in stop/scroll mode, this button can be used to step backward through events one by one in the animation.

Animator mode selection The animator can be in either explore or scenario mode. The user can do this selection in the menu. In the status bar the currently selected mode is highlighted and in scenario mode a small window for controlling the scenario and recording a play list is presented. When this window is closed, MainFrame minimizes it to the status bar as a button for restoring the window.

Layer selection The user can select the layer that is currently shown by clicking the button of the layer he wants to see either in the menu or on the tool bar. When such a button is pushed, all animation panels are recreated with correct settings.

Language selection When the user selects a new language, the whole user interface is recreated using the strings from the selected language. This is done by calling UserInterface method recreateUI.

Help The help command opens a new window with an HTML help document shown in it.

About The about command opens a window with tabs for general information, licence information and information on the team behind the product.

5 Animation panels

All the animation panels that can be placed on the four animation panel spaces in the animator have to extend the abstract class AbstractAnimationPanel. This class handles some aspects of the communication between the panels and the MainFrame. It also implements the interface ControlSignalListener that allows the panels to receive control signals from the Control Signal Framework.

In this version of the animator four animation panels are created:

- Message Sequence Chart
- Encapsulation
- Unit Flow Orchestration
- Notes

In addition to these four main animation panels there is a fifth AbstractAnimationPanel, TimePanel that is used only to display the current animation time in the status bar.

The animation panels are described in more detail in the following subsections.

5.1 Message Sequence Chart

The Message Sequence Chart (MSC) animation type is the primary animation type for the DaCoPAn animator. In Figure 6 a sketch of the animation panel is presented. It is then explained in the further subsections.

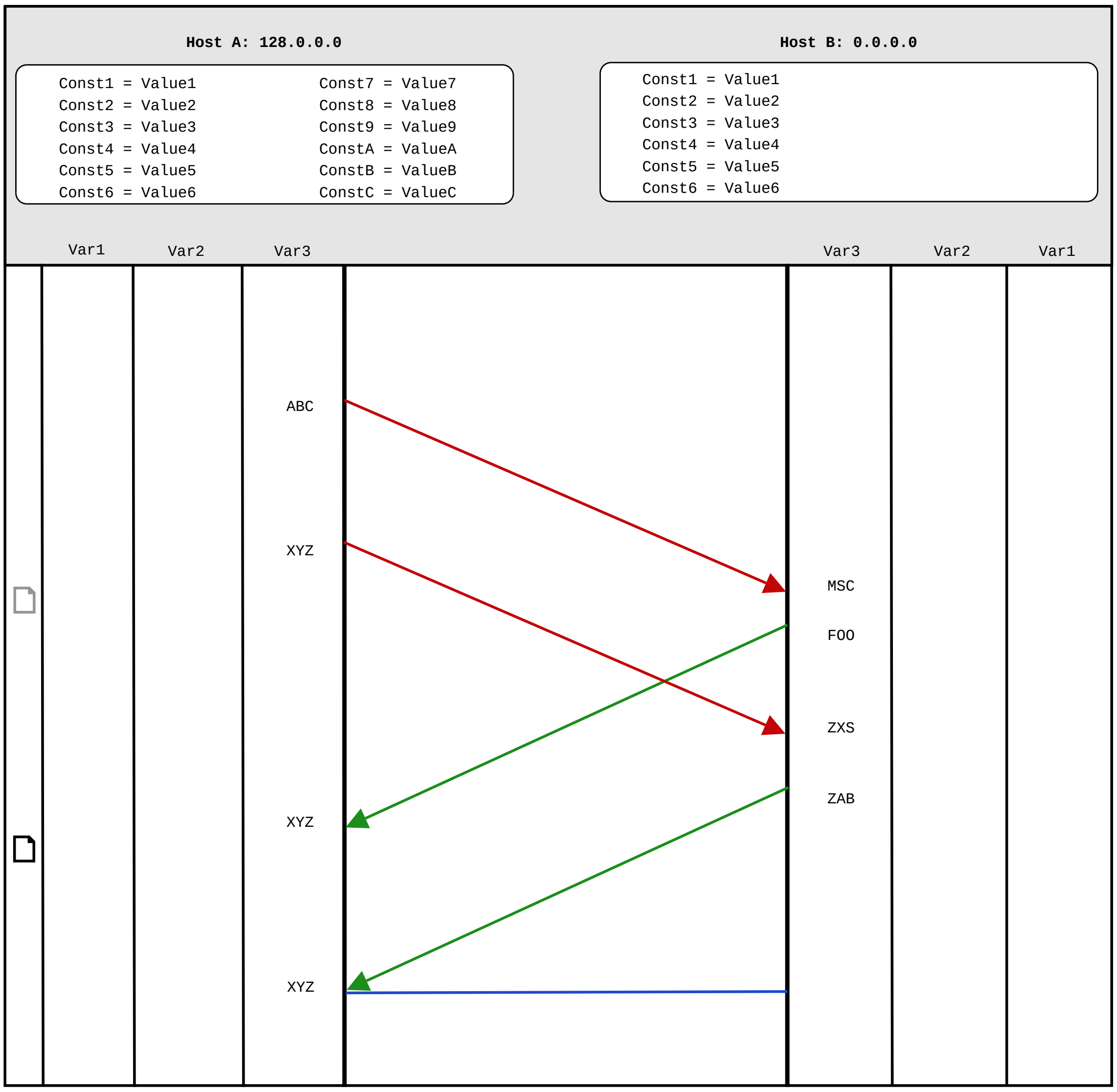


Figure 6: MSC panel

5.1.1 Columns

The MSC presents symbolic information (text and numbers) in vertical columns. Each column shows information about exactly one host variable. The selection and number of columns is configurable through the MSC settings dialog. The order of columns is not configurable. Note that only values of host variables will be shown in columns (and values of protocol variables will be shown in the drawing area).

The columns are mirrored for both of the hosts so that exactly the same variables are shown for both hosts. The showing order of columns is also mirrored, so that the column closes to drawing area always presents values of the same variables on both sides of the drawing area.

The width of the columns is determined by the maximum length (in characters) of the values in that column. This information can be queried from the data structure classes.

5.1.2 Notes

The presense of a note is signaled by showing a small icon in a notes column that is always present as the left-most column next to the left edge of the component. (That column is never mirrored, as there is only one notes column.) The top edge of the icon corresponds to the exact location of the note on the time axis.

5.1.3 Host constants

Host constants are values related to a host that don't change according to networking events. Host constants are shown at the top portion of the panel, right below the name labels of the hosts.

5.2 Encapsulation

The encapsulation animation occupies the main frame when it is activated. The UFO panel is cleared and disabled, and the note panel displays the note of the specific ENC animation. The tool bar rewind and fast forward functions are disabled.

5.2.1 Note framework for ENC

Each ENC animation can contain a single note. The note is bound to the TransferUnit of which the ENC diagram is constructed from. The note can be retrieved from NoteManager by calling the method getEncNote(TransferUnit) with the active ENC unit as the parameter.

5.2.2 The ENC tree model

The generation of the encapsulation tree for an ENC animation is the responsibility of a class called **ENCTreeModel**. This class contains all the logic needed to select the most interesting units in the encapsulation tree of a selected unit for display in the animation. By default the tree model produces ENC trees with a maximum of three units on the same level, but this value can be changed to any other interger value.

5.2.3 General layout

The ENC panel consists of three areas, corresponding to the different layers in the communication. The application layer can contain up to one unit, and the transport and network layers can be populated with up to three units. If the application layer contains no unit (this corresponds to showing protocol exchanges where the application level protocol is not recognized/supported) then the transport layer, being the highest layer, contains exactly one unit. However, the design of the encapsulation animation is flexible enough to support layers other than the three usually present in scenarios. As an example of a three-layer encapsulation diagram, see figure 7.

A unit contains a fixed width field variable area for displaying unit variable values. The rest of the available width is used for a payload area. Both sections of the unit are marked with small numbers to indicate their sizes. The physical widths of the areas are not necessarily in relative proportion. On lower layers lines are draw to designate the composition of the unit, the lines are connected to a roughly corresponding location of the encapsulating unit. See diagram 8.

When the application level protocol is not recognized/supported by DaCoPAn, the application layer area is rendered by drawing a cloud with a question mark in it, or some other symbolic representation of the fact that the protocol is not known, therefore the encapsulation can not be shown. See diagram 9.

At the bottom of the ENC panel there are three buttons used for initiating either encapsulation or deencapsulation animations, and returning to the MSC mode to continue with the scenario.

5.2.4 Animating the ENC diagram

Draw one layer at a time, progress without stopping, at the end you have the full diagram. DeENC is the same with reverse layer ordering. In between layers animate the lines to be drawn gradually from bottom to top or vice versa, depending on the direction.

5.3 Unit Flow Orchestration

The UFO panel is designed to visualize the message sequence data as a continuos animated flow of units moving between hosts. It has two channel areas for animating the

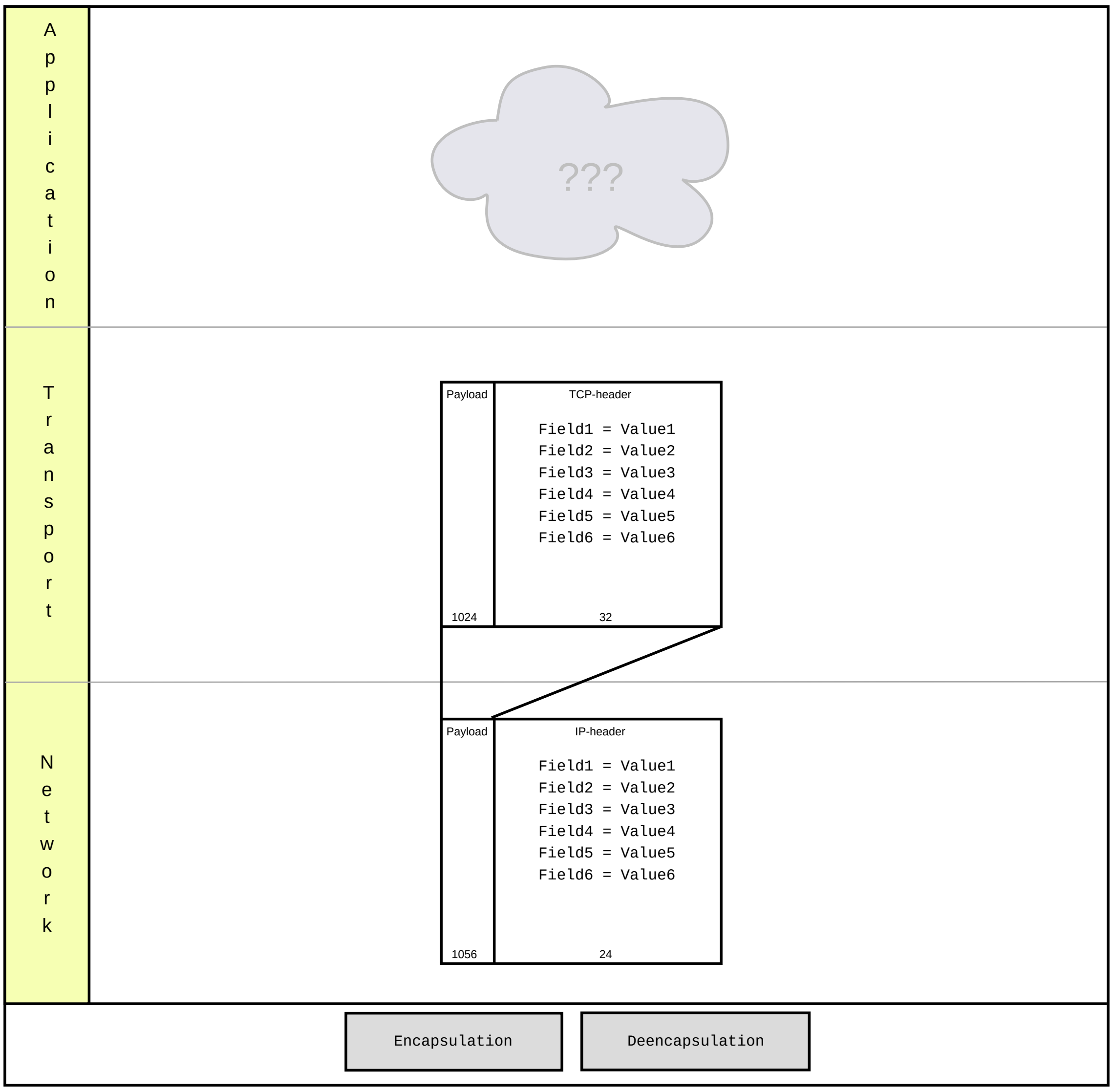


Figure 7: Application level protocol unknown, TCP on transport level

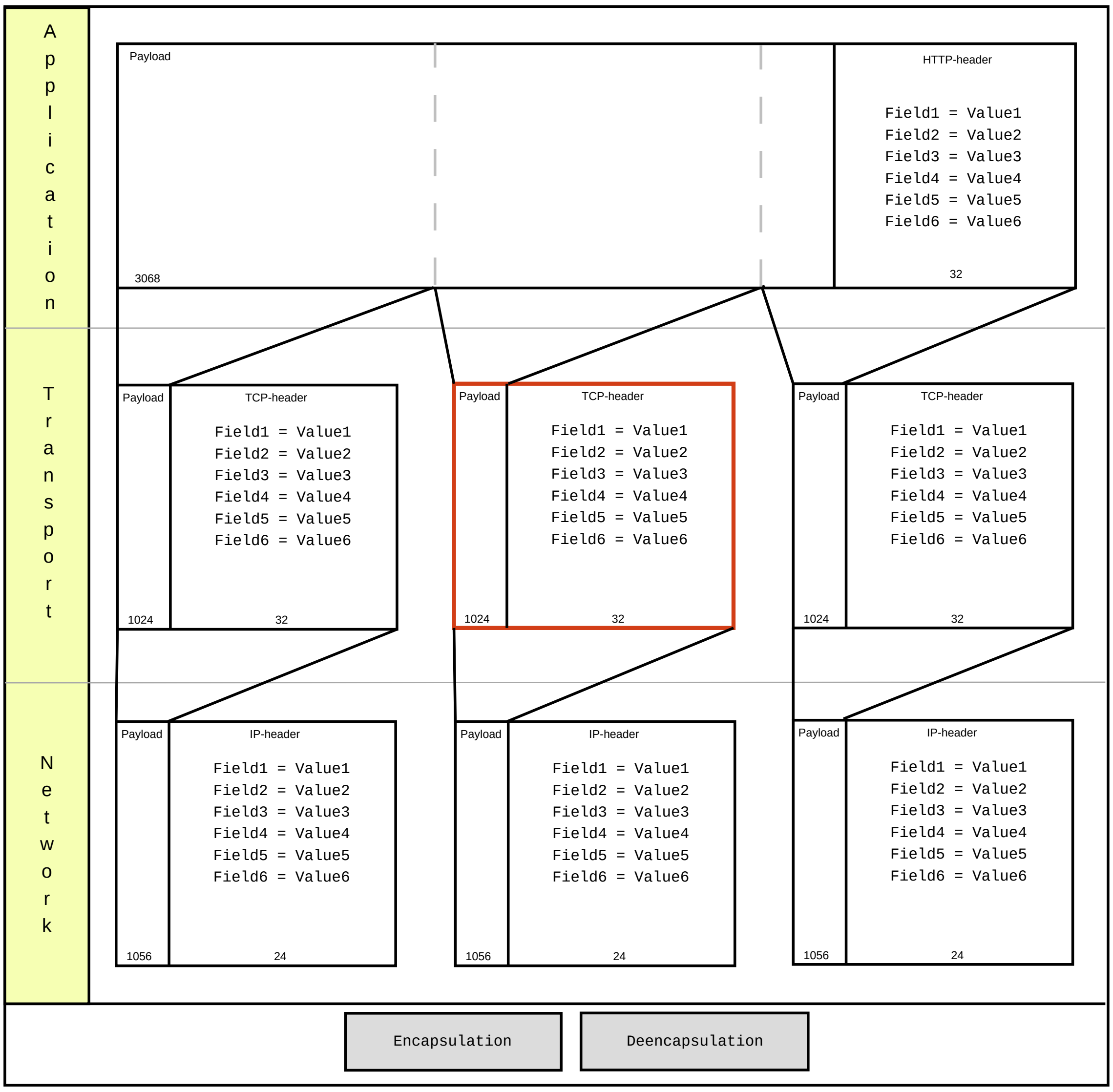


Figure 8: ENC panel when the application level protocol is known

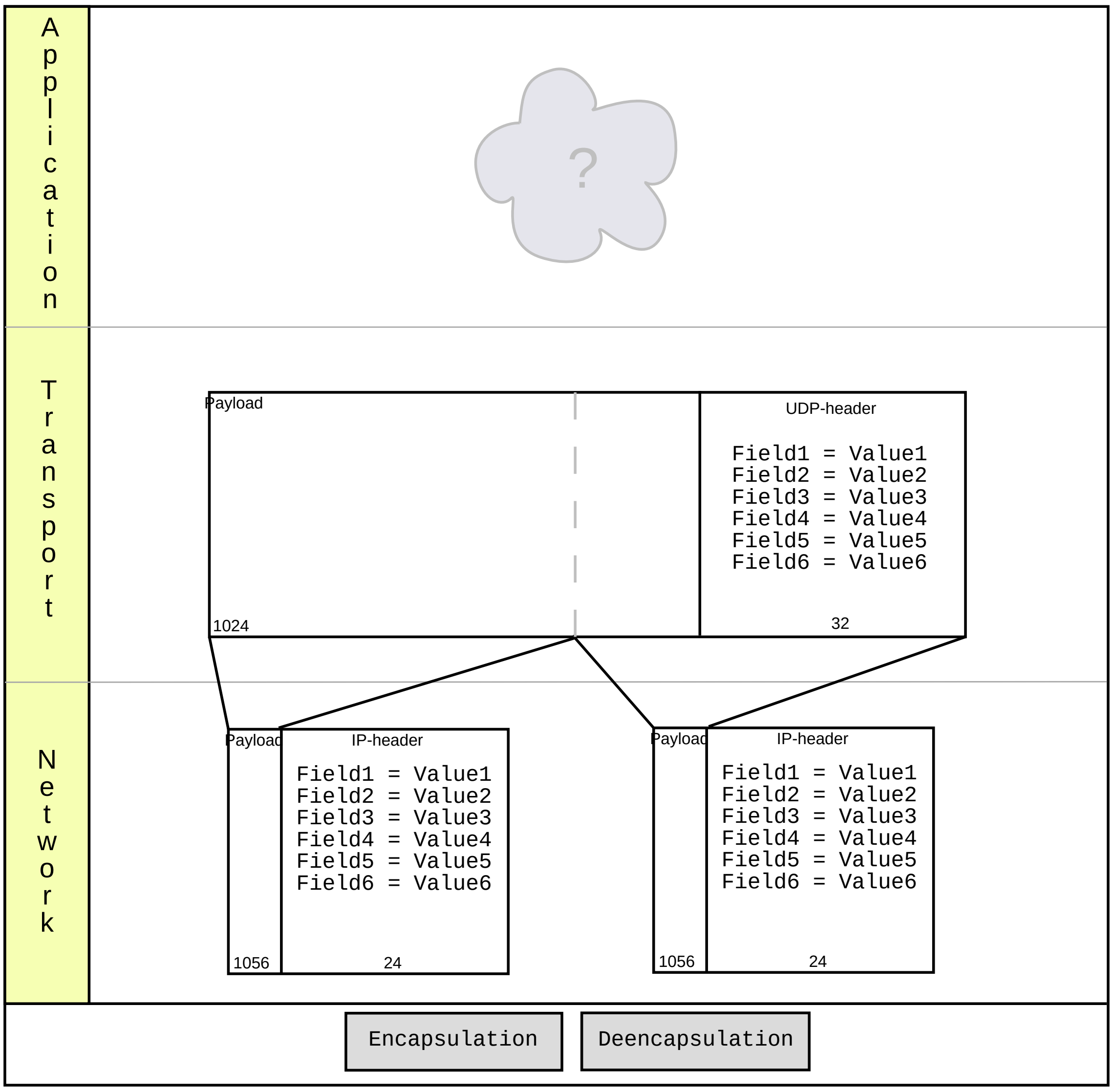


Figure 9: UDP on transport level

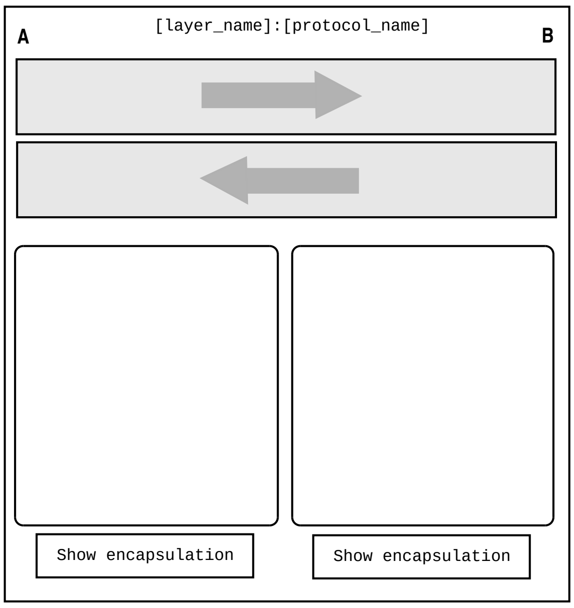


Figure 10: Empty UFO panel, e.g. displayed during ENC

movement of the units and two unit variable fields for showing the data that is contained by the selected transfered units, one for each direction. See figure 10 for the main layout of the UFO panel.

The UFO panel is a subclass of AbstractAnimationPanel which in turn implements the ControlSignalListener interface. This means that it has the method advance() in which it is asked to update to a new animation state. When the UFO panel is drawn, it calculates the position for each unit and draws them to the correct place on the appropriate channel.

The data of the active units is shown in the data areas. When a new unit appears to the visible area, it is selected as the active unit, but the user can also select a unit to be activated by clicking on it. The selected unit stays selected until some other unit is activated, or the unit is received. Units are activated automatically when they are created, so that the latest unit is always the default active unit.

Another important use for the UFO panel is that it can be used to move into encapsulation mode. There are two buttons under the data fields that can be used to show the encapsulation for the selected unit. When a unit is active, this button can be clicked. The main frame switches to showing encapsulation instead of MSC animation.

In paused mode the UFO automatically draws lines from the field value boxes to the packets in question. These lines are erased when playing is resumed. See figure 12.

If all unit fields don't fit in the reserved text areas, the text areas should scroll downward. For clarity's sake, each unit field should occupy it's own row. See figure 11

The 2 channels (pipes) contain only one color of units, which have a darker border when they are activated. The relative sizes of the units is shown by unit length. The colors used for hosts A and B can be green and red, but other colors may prove to be superior. Dropped packets are marked with a red cross, and disappear when they arrive at the center

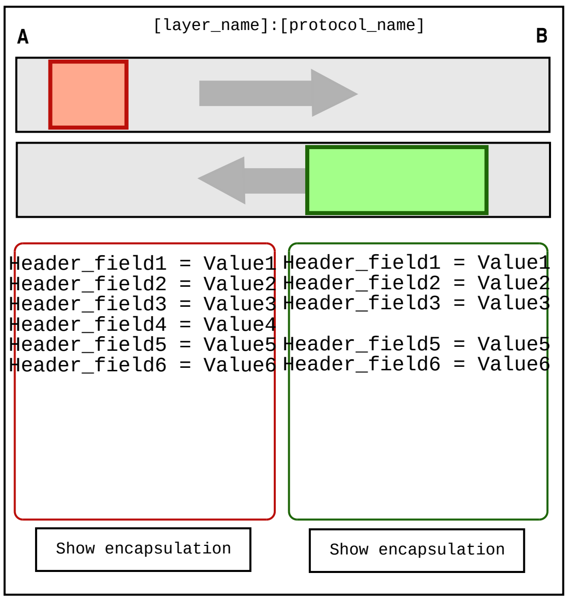


Figure 11: UFO panel in play mode

of the channel.

6 Animation settings

The animator has to keep track of the settings that the user has chosen to visualize the different events contained in the protocol events file. The general settings represent a configuration that will be applied to some different features of the animator.

6.1 Generic animation settings

There are some generic animation settings that are not related to any individual animation type. These settings will be represented in the class GeneralSettings. These values are related with settings that affect to the performance of the animation like the refresh rate for the animations and the antialising options for the different animations.

These settings are:

- **Refresh delay:** Sets the number of milliseconds between two refreshes of the animation. Increasing its value means that the screen will be refreshed few times in a second, the perception of the user is that the animations are less smooth in the movement, but the performance of the animation is increased. If the values are decreased the performance is worst but the element in the animation move softer.

- **Anti-Alias:** Sets the smoothness of the elements drawn in the different screens. If the checkbox is selected the elements in the animations are drawn smoother, but in

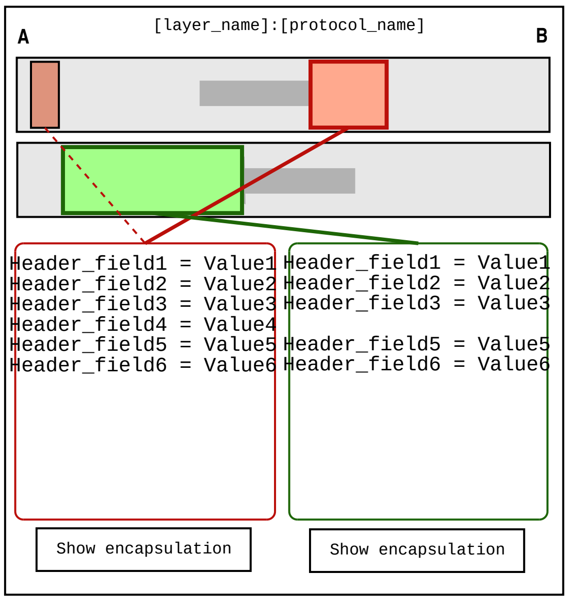


Figure 12: UFO panel in stop/scroll mode, with altered selection

the other hand the performance of the animator is worse. De-selecting it we get the inverse results, poor graphics but better performance.

A basic class diagram for GeneralSetings is in the Figure 13

6.2 MSC settings

There is a separate settings dialog and class for configuring MSC settings. At least the following items need to be configured:

- **Display settings**
  - Columns: the selection of columns out of the possible host variables.
  - Drawing area variables: the protocol variables that are drawn along the packet lines in the drawing area.
- **Scale settings**
  - Scale mode: Defines a factor of scale to draw the MSC animation. (for instance: Seconds to seconds: one second of network time is animated in one second).
  - Visual Scale: Adjusts the number of pixels used in the MSC animation to draw one unit of network time.
  - Time Scale: Number of seconds that takes to the animation to show one unit of network time. Higher values mean that the animation will move slower.

To fulfill all these needs for the animator a class SettingsMSC should be implemented containing: Lists for the variables and header fields selected to be drawn in the animation. Attributes to adjust the scale settings and the panels (scale mode, visual scale, time scale). Also an attribute related directly with the scenario mode should be added here, and autoplay boolean attribute to set wether the next element in a scenario playlist should be played after the previous one finishes. Another couple of attributes to add to the MSC setting that come from the scenario playlist needs are the start and end time in its items. A diagram for this SettingsMSC class is represented in the Figure 13

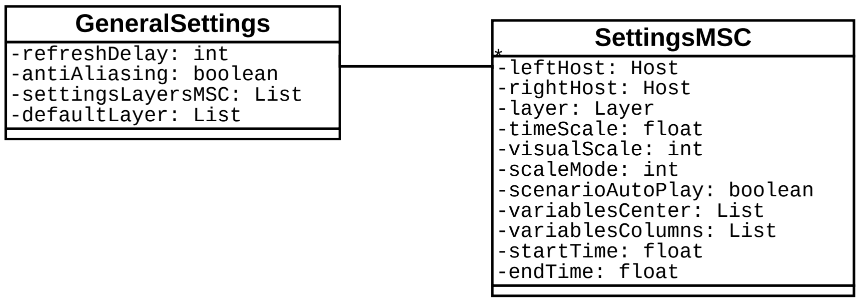


Figure 13: Classes for storing general settings and settings for the MSC animation

6.3 Settings Dialog

The user can access the settings using a dialog. Within that dialog the settings can be grouped in several panels applying the following criterion:

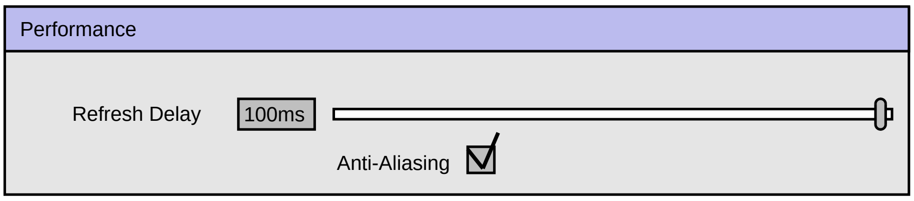
- Performance settings: will contain the values that provide the class GeneralSettings. A sketch for this panel is drawn in the Figure 14.
- Scale settings: will contain the selection of scale mode and the visual and time scale values. A sketch for this panel is drawn in the Figure 15.
- MSC related settings: will show the lists of variables and header fields that can be selected to be visualized in the MSC animation and other settings related with MSC. A sketch for this panel is drawn in the Figure 16.

Sketches for the settings dialog panels:

6.4 Settings for the different animation modes

The settings dialog has to be adapted to the different animation modes, allowing to change only the setting classes values specific to the active working mode.

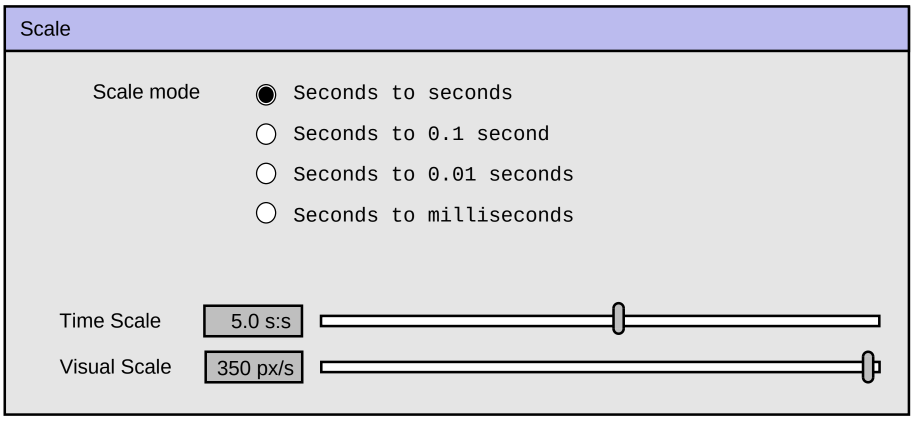
- Explore mode: the user should be able to modify the performance settings, scale settings for all the layers in the animation and the variables and header fields to include in the animation for every existing layer in the animation.


Performance

Refresh Delay 100ms

Anti-Aliasing ✓

Figure 14: Performance settings panel

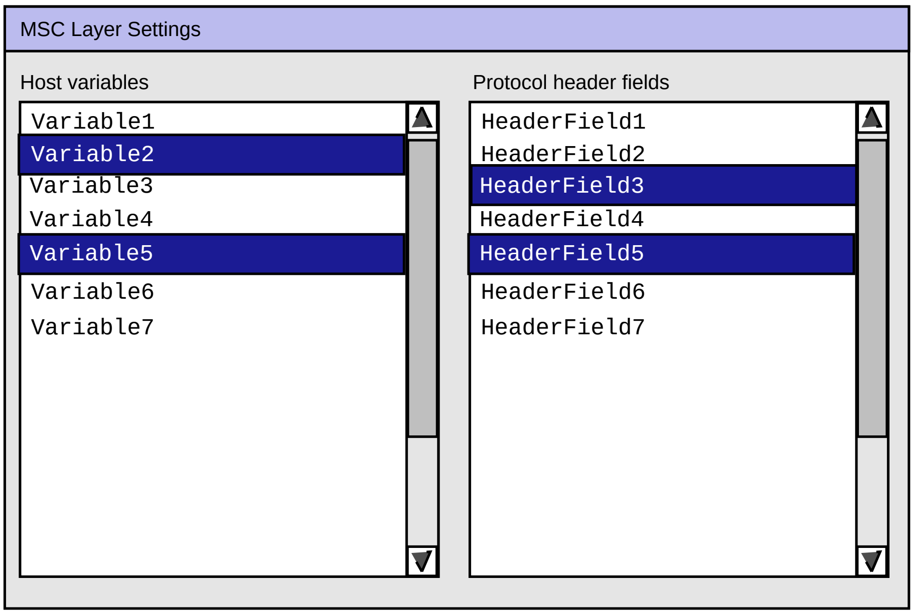

Scale

Scale mode
● `Seconds to seconds`
○ `Seconds to 0.1 second`
○ `Seconds to 0.01 seconds`
○ `Seconds to milliseconds`

Time Scale 5.0 s:s

Visual Scale 350 px/s

Figure 15: Scale settings panel


MSC Layer Settings

Host variables	Protocol header fields
Variable1	HeaderField1
Variable2	HeaderField2
Variable3	HeaderField3
Variable4	HeaderField4
Variable5	HeaderField5
Variable6	HeaderField6
Variable7	HeaderField7

Figure 16: MSC settings panel

- Scenario mode: due to the purpose of creating a item for the scenario playlist, a few settings can be tuned. These are scale settings and the header fields and variables, all of them should be modified only in the active layer. In this working mode also should be included in the panels a checkbox to set the autoplay feature in the scenario playlist items.

7 Control signal framework

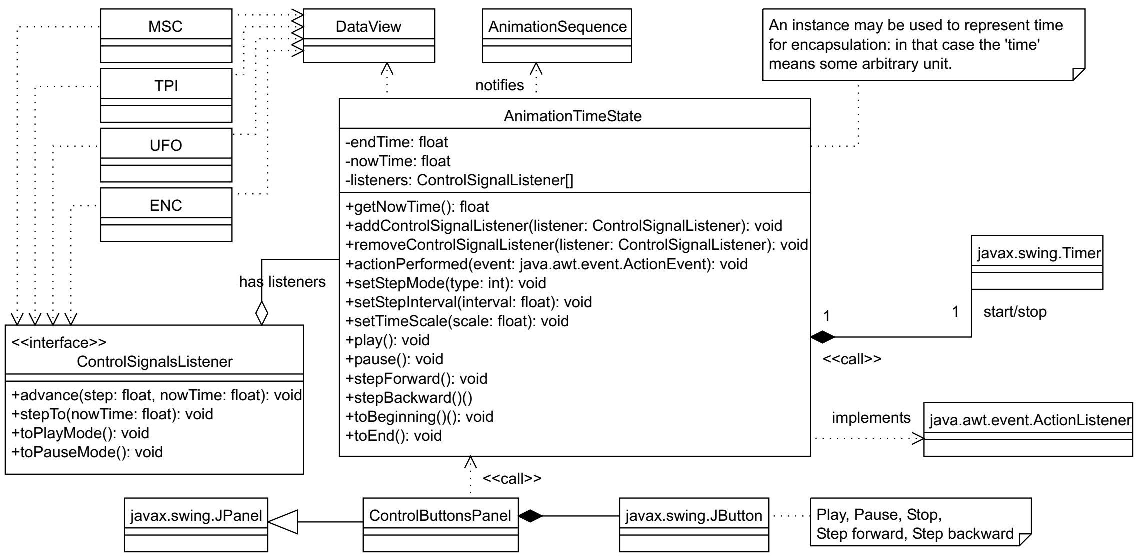

MSC
TPI
UFO
ENC
DataView
AnimationSequence
An instance may be used to represent time for encapsulation: in that case the 'time' means some arbitrary unit.
notifies
AnimationTimeState
-endTime: float
-nowTime: float
-listeners: ControlSignalListener[]
+getNowTime(): float
+addControlSignalListener(listener: ControlSignalListener): void
+removeControlSignalListener(listener: ControlSignalListener): void
+actionPerformed(event: java.awt.event.ActionEvent): void
+setStepMode(type: int): void
+setStepInterval(interval: float): void
+setTimeScale(scale: float): void
+play(): void
+pause(): void
+stepForward(): void
+stepBackward()()
+toBeginning()(): void
+toEnd(): void
has listeners
<<interface>>
ControlSignalsListener
+advance(step: float, nowTime: float): void
+stepTo(nowTime: float): void
+toPlayMode(): void
+toPauseMode(): void
javax.swing.Timer
1
1
start/stop
<<call>>
implements
java.awt.event.ActionListener
<<call>>
javax.swing.JPanel
ControlButtonsPanel
javax.swing.JButton
Play, Pause, Stop,
Step forward, Step backward

Figure 17: Control signals framework

7.1 AnimationTimeState class

The central class of the control signals framework. The main functionalities are:

- Receive user signals as method calls (play(), pause(), etc.). The calls are made by the buttons panel component.
- Make calls to all registered ControlSignalListeners according to either user signals (in pause mode) or timer events (in play mode).

Methods play(), pause(), stepForward(), stepBackward(), toBeginning(), toEnd(): These methods are called by the toolbar buttons. AnimationTimeState reacts to these calls by making necessary state changes and necessary calls to ControlSignalListeners.

Methods addControlSignalListener, removeControlSignalListener: Add or removes a ControlSignalListener. Control signal listeners receive coordinated events for the animation, like animation 'ticks'.

Method setStepMode: Used to select the desired step mode from possibilities ’time slice’, ’host A events’, ’host B events’, ’send events’, ’receive events’ and ’all events’.

Method setStepInterval: The desired step interval in physical computer clock milliseconds for the ’ticks’ in play mode.

Method setTimeScale: Sets the time scale of the animation. Scale of 1.000 means presenting one second of network exchange takes ONE second of real time and scale 10.000 means presenting one second of network exchange takes TEN seconds of real time.

Other things to note:

- An AnimationTimeState object contains a javax.swing.Timer instance. That is used to get timed animation ’ticks’ to run the animation in play mode. For each timer event the AnimationTimeState object gets a call on its own actionPerformed method, does the required accounting and makes subsequent calls to all listeners.

- For all other step modes than the ’time slice’, the DataView is queried for the size of the next tick.

- When run in scenario mode (for presenting an animation sequence, as opposed to explore mode) the AnimationTimeState needs to contain an **endTime**. When the endTime is reached, the AnimatioTimeState puts itself in pause mode and calls the showNext method of the AnimationSequence object. This is a signal for the AnimationSequence to start presenting the next item in the animation sequence.

- AnimationTimeState takes into account the actual processing rate of the timer events. In times when the CPU load reaches 100target rate. In these cases the AnimationTimeState adjusts the logical step size to reflect the actual processing speed. The logical step size means the step expressed as network time in the advance(step, nowTime) method). Because of this mechanism, the actual time spent in playing an animation from beginning to end depends ONLY on the length of the protocol event data and the time scale, NOT on computer speed (within reasonable accuracy). However, the number of animation frames drawn during the presentation of the animation MAY depend on computer speed.

7.2 ControlSignalsListener interface

All animation panels need to implement the ControlSignalsListener interface. The main UI needs to register all visible animation panels as ControlSignalsListeners for the current AnimationTimeState object.

Method advance(step, nowTime): tells the animation panel to make an incremental advance from previous state to nowTime. The difference between previously shown time state and nowTime is given in parameter step. If the animation panel doesn’t need to make difference between advance operations and stepTo operations, this method can be implemented as just "public void advance(float step, float nowTime) stepTo(nowTime); ".

It is the responsibility of the animation panel to call repaint() on itself after it has updated its state.

Method stepTo(nowTime): tells the animation panel to show the animation time given in parameter nowTime, regardless of the previously shown state.

Method toPlayMode(): tells the animation panel that a continuing sequence of advance(step, nowTime) calls will probably follow. (Some panels may wish to disable scrolling mode as result of this call.)

toPauseMode(): tells the animation panel that the following stepTo-calls will be results of direct user input. (Some panels may wish to enable scrolling mode as result of this call.)

8 Animation sequence framework

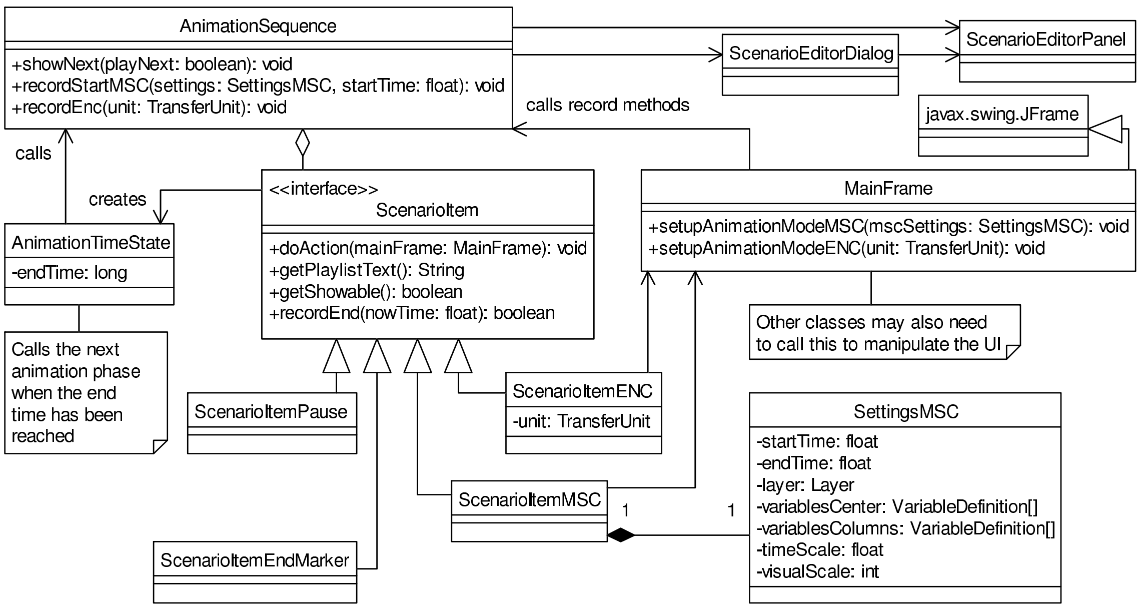

AnimationSequence
+showNext(playNext: boolean): void
+recordStartMSC(settings: SettingsMSC, startTime: float): void
+recordEnc(unit: TransferUnit): void
ScenarioEditorDialog
ScenarioEditorPanel
calls record methods
javax.swing.JFrame
calls
creates
<<interface>> ScenarioItem
+doAction(mainFrame: MainFrame): void
+getPlaylistText(): String
+getShowable(): boolean
+recordEnd(nowTime: float): boolean
MainFrame
+setupAnimationModeMSC(mscSettings: SettingsMSC): void
+setupAnimationModeENC(unit: TransferUnit): void
AnimationTimeState
-endTime: long
Other classes may also need to call this to manipulate the UI
Calls the next animation phase when the end time has been reached
ScenarioItemPause
ScenarioItemENC
-unit: TransferUnit
SettingsMSC
-startTime: float
-endTime: float
-layer: Layer
-variablesCenter: VariableDefinition[]
-variablesColumns: VariableDefinition[]
-timeScale: float
-visualScale: int
ScenarioItemMSC
1
1
ScenarioItemEndMarker

Figure 18: Animation sequence framework

Animation sequence framework provides a way for storing a sequence of individual presentation types, and ways for creating such sequences. The main idea is a "playlist" type of list of individual presentations. These presentations may currently be MSC animations (with or without auxiliary animation types), Encapsulation animations, BreakPoints or pauses.

8.1 AnimationSequence class

AnimationSequence class actually contains the list of presentations. For each item in the list it:

- Creates an instance of AnimationTimeState according to the requirements of the item to be presented. (e.g. sets a suitable endTime)
- Calls the main UI to set the desired view mode. Gives the newly created AnimationTimeState instance as parameter to main UI, so that it can be registered as listener for control button signals.
- Receives a call from the AnimationTimeState instance when the endTime is reached, and initiates the presentation of the next item in the list.

8.2 ScenarioItem interface

An interface that all the items in the animation sequence need to implement. The interface functionality is related to providing a way to call for the item to carry out its ’action’ and to have it give the necessary information about itself for e.g. presentation in the scenario play list.

8.3 ScenarioItemMSC class

Initiates showing an MSC animation. Contains a reference to SettingsMSC objet which contains settings specific to the MSC animation panel.

Any settings in SettingMSC or other settings objects are created and edited through specific editing dialogs invoked through the main UI.

The semantics for the start time and end time for an MSC scenario item are as follows: when the item is selected from the scenario play list, the current AnimationTimeState instance is set to point to the **start time** of the item. When the animation is played past the **end time** of the item, the AnimationTimeState will call the AnimationSequence to select the next item in the play list, and thus end the presentation for the previous item.

8.4 ScenarioItemENC class

Instructs the MainFrame to show the encapsulation for the designated TransferUnit.

8.5 ScenarioItemPause class

An item in the playlist that can be used to wait for user input before continuing.

8.6 BreakPoint class

Can be used as an item in the animation sequence. Will show a dialog and wait for user input before continuing the sequence.

8.7 Settings objects

The settings objects for animation panels, edited through specific editing dialogs and used by their respective animation panels. Currently the only actual settings object is SettingsMSC.

8.8 ScenarioEditorDialog and ScenarioEditorPanel classes

The dialog class in related only to presenting the editing widgets inside a dialog, and how the dialog itself behaves with the MainFrame. The actual editing functionality is implemented as a separate panel to make it general so it can be used in the UI in other ways than in its own dialog. For all editing functionality, the panel class and the AnimationSequence class communicate directly with each other.

The ScenarioEditorPanel presents the scenario play list in an interactive list component. All changes from editing will be instantly visible in the list. At all times the list can be used to select the desired item from the list for playing or editing. The ScenarioEditorPanel also contains the necessary buttons for inserting, deleting and editing scenario items.

8.9 Recording an animation sequence

Recording an animation sequence happens as an interaction between an AnimationSequence object and the main UI. When the user wants to start recording a new animation sequence, he selects 'Scenario mode' command from the 'Animation' menu. At that time the ScenarioEditorDialog is shown, with the ScenarioEditorPanel as its contents. The dialog is non-modal, so that it is possible to operate all UI widgets while it is visible. The recording mode is toggled on and off by using the 'Recording mode' toggle button in the ScenarioEditorPanel.

When the recording mode is on, whenever the user presses play, the main UI makes a method call to the recordStart method of AnimationSequence. That causes the AnimationSequence to create a new ScenarioItem object. The exact type and settings correspond to the active mode and settings in the main UI. The created AnimationAction object is otherwise complete, but (in case of actions that do have a duration) it lacks the endtime.

Whenever the user changes animation settings, layer or view mode, the main UI makes method call recordEnd to the AnimationSequence. The next recordEnd after a recordStart call is considered the end of the item being recorded. When the recordStop is called, the AnimationSequence is able to complete the ScenarioItem with the endTime information, and insert the complete object in the animation sequence list.

A summary of responsibilities for different classes regarding recording an animation sequence:

Main UI:

- Receive user input events about starting the recording, play events, mode switches etc.
- Keep track of recording mode (on/off).
- Signal play events to AnimationSequence (params: start time, settings).
- Signal mode switch to AnimationSequence (params: end time).

AnimationSequence:

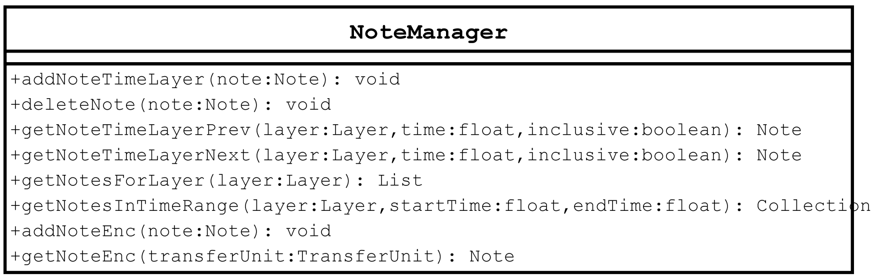
- Receive 'play' and 'mode switch' events from main UI.
- Create ScenarioItem objects.
- Insert ScenarioItem objects in the list that makes up the animation sequence.
- Update information in ScenarioEditorPanel.

9 Notes framework

In this section we will present a framework to work with the notes that will be shown in the different types of animation to increase the educational abilities of the software.

There are two main types of notes to show that correspond to the Message Sequence Chart (MSC) animation and Encapsulation (ENC) animation. Both of them have different purposes and they will be designed to fulfill the different needs that can appear for this types of animation.

The classes for both types of notes are included into the data structures due to some associations that need these classes with other ones present in the data structures to make easier the access to all this data to the control signals framework through the NoteManager (see Figure 19). Any notes added to the NoteManager are stored along with other animation scenario data, so they're persistent.



```
NoteManager

+addNoteTimeLayer(note:Note): void
+deleteNote(note:Note): void
+getNoteTimeLayerPrev(layer:Layer,time:float,inclusive:boolean): Note
+getNoteTimeLayerNext(layer:Layer,time:float,inclusive:boolean): Note
+getNotesForLayer(layer:Layer): List
+getNotesInTimeRange(layer:Layer,startTime:float,endTime:float): Collection
+addNoteEnc(note:Note): void
+getNoteEnc(transferUnit:TransferUnit): Note
```

Figure 19: Interface to access notes in the data structures

9.1 MSC Notes:

These notes represent an action that happens in a specific layer in a specific moment in time. Then we only need to map this information into the class that will represent the MSC notes, but this class also aims to serve as note for any other Time-Layer event in an animation. An identifier will also be added into it to make easier referring to the notes to edit and modify them and locating them more efficiently.

In the NoteManager that is used to access to the data contained in the data structures has to contain the following methods to manage the adding, editing and deleting of the notes, and fetching them as well.

addNoteTimeLayer (note:Note):void
Add a note for a specific layer in a specific moment of time. In case that the note already exists in that layer and time the note should be edited and the text replaced with the new value.

deleteNote(note:Note)
- Deletes the given note from the manager.

getNotesTimeLayerPrev (layer:Layer, time:float, inclusive:boolean): Note
- Returns the note on the given layer that's located before the given time.

getNotesTimeLayerNext (layer:Layer, time:float, inclusive:boolean): Note
- Returns the note on the given layer that's located after the given time.

getNotesForLayer (layer:Layer): List
- Returns a list with all the notes present in the specified layer passed as parameter. This has a similar function as getUnits has with TransferUnit

getNotesInTimeRange(layer:Layer, start:float, end:float): Collection
- Returns a list with all the notes present in the specified layer passed as parameter in an interval between the start and end times.

9.2 ENC Notes.

The notes for encapsulation have to be represented in a different manner than the notes for MSC due to the different purpose of the representation that will have in the animation. Thus, the encapsulation notes can have one note per encapsulation tree and this note is shown at the end or during the ENC animation.

This note for encapsulation should be in a different class as is information contained in the data structures, but represents a different concept than the information coming from the protocol events and must be separated from them.

The methods to use the Encapsulation notes that are added to the NoteManager are:

addNoteENC (note:Note)
- Adds the note (that's specific to a TransferUnit) to the manager. - If already exists a note in the specified unit the existing text will be replaced with the one provided as parameter. (edit function).

getNoteEnc(transferUnit:TransferUnit):Note
- Returns any encapsulation -specific note of the given unit.

10 Protocol events file reader

The protocol events file (PEF) is the interface between the Analyzer and the Animator components. The Animator contains a customized reader whose purpose is to read in the protocol events data and create a corresponding presentation of the data as instances of the Animator's internal data structure classes.

Even though the format of choice for the PEF is XML (eXtensible Markup Language) at the moment, the functionality of the PEF reader is abstracted to an interface called **ProtocolEventsReader**, of which the **XMLProtocolEventsReader** is only one possible implementation: future versions may provide other implementations, should more suitable options be found

10.1 ProtocolEventsReader interface

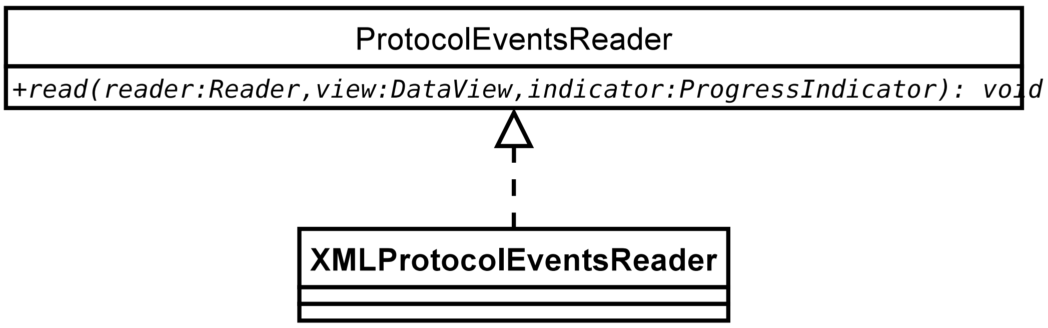

ProtocolEventsReader
+read(reader:Reader,view:DataView,indicator:ProgressIndicator): void
XMLProtocolEventsReader

Figure 20: Class diagram for the protocol events reader

The ProtocolEventsReader interface specifies a single method (**read()**) that is used to process a PEF. It accepts three parameters, a Reader that the PEF will be read from,a DataView object that the data in the PEF will be passed to and a ProgressIndicator for indicating progress to the user interface. That is, the ProtocolEventsReader uses a callback mechanism: whenever it encounters a piece of data in the PEF, it constructs the Java object that corresponds to the data. For example, if the reader reads in an XML element containing information on a host, it constructs the corresponding Host object and calls the addHost(host) -method on the DataView instance.

Using a callback mechanism instead of reading all the data in first and then providing getter methods for it (e.g. getHosts(), getFlows()) in the ProtocolEventsReader interface has many advantages: firstly, the data doesn't need to be stored in the reader itself, instead it can be passed to the DataView instantaneously. Secondly, having addXXX() -methods in the DataView interface makes writing unit tests easier, as the view can be easily populated with test data.

See figure 21 for an example where the reader first encounters a host, then a transfer unit and adds them to the DataView instance.

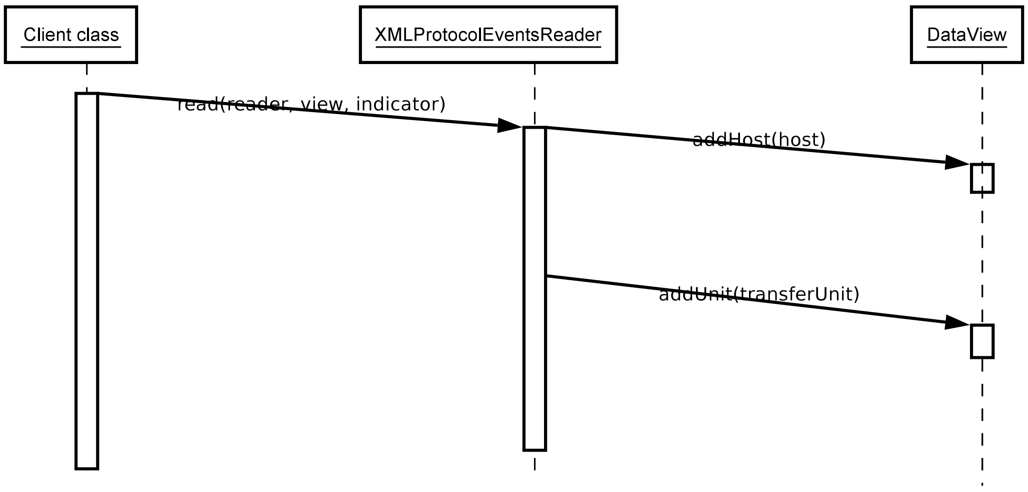


Figure 21: Sequence diagram for PEF reader

10.2 ProgressIndicator interface

As loading a large PEF file can easily take a long while, it's preferable to indicate the progress of loading to the user. A special interface, `ProgressIndicator`, is defined for this purpose. The PEF reader can signal any classes implementing this interface when any progress is made, and these classes can in turn update a progress monitor in the user interface, for example.

10.3 XML protocol events reader

10.3.1 XML support in Java 1.4

The Java 2 platform versions 1.4 and up (which is required by the Animator) provide good facilities for processing XML without the need for any external libraries: namely, the JAXP (Java Api for XML Processing). An XML parser is included in the standard libraries along with support for the two most common XML parsing APIs: DOM (Document Object Model) and SAX (Simple API for XML). The main difference between the two APIs is that DOM creates an in-memory object presentation of the XML tree while SAX simply produces events for any XML elements it encounters. Thus DOM uses more memory while SAX is more gentle on system resources: on the other hand, using the DOM API requires less effort. As the performance of reading in the protocol events data is not an issue for the Animator (it is only done once for every scenario), DOM should be used to make the code for the reader as clear and concise as possible.

10.3.2 Implementation using the JAXP classes

The XML-based events reader can obtain a DOM parser using the class **javax.xml.parsers.DocumentBuilderFactory**. The parser can then be used to parse the PEF to an in-memory XML tree, which can then be processed to create instances of the internal data structures of the Animator.

Classes for accessing the actual DOM XML tree are located in a package called **org.w3c.dom**. For example, each element (tag) in the document is represented as a **org.w3c.dom.Element** in the object tree.

11 Scenario data

In addition to the protocol events data, each networking scenario that's visualized using the Animator can contain additional data that's used to set the scenario presentation up: this data includes animation panel settings, notes and animation sequence data. This additional information needs to be saved along with the protocol events data so that a scenario doesn't need to be set up every time it is opened using the Animator.

11.1 Scenario file

All the information is contained in a single file (the **scenario file**) so that viewing a scenario doesn't require the user to download multiple files from the course homepage (for example). The scenario file is implemented as a Jar archive that allows multiple files to be stored within one file (the Java platform has built-in support for both Zip and Jar archives). See figure 22 for an example.

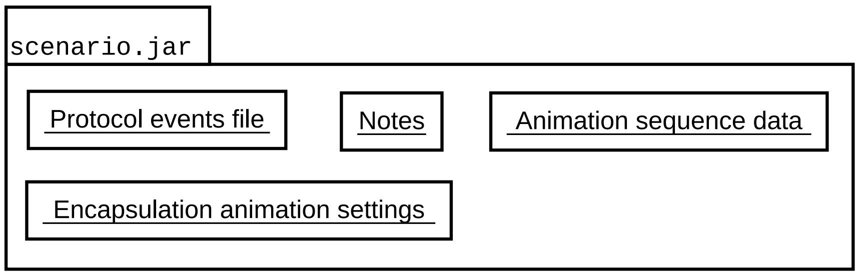


Figure 22: Scenario archive file contents

The Animator will be able to read both plain protocol events files and scenario files. When the animation sequence is altered, the user has the option of saving the protocol events (that remain unchanged) along with the settings data to a separate scenario file.

11.2 Saving and loading the scenario data objects

Scenario data (notes, settings etc.) is only of interest to the Animator itself, i.e. it is not necessary for any other party (such as the Analyzer) to read it. Thus the format in which the data is stored is not as important as with the protocol events data. This in turn allows the Animator to take advantage of existing persistence mechanisms for objects: for example, instead of specifying a XML format for notes and writing special classes for saving and loading them, the Animator could simply use standard Java serialization facilities to save and load the data.

However, the standard Java serialization procedure is prone to class format incompability errors: if a settings class is modified, for example, older serialized instances of that class might not be readable anymore. Also, serialized objects are stored in a binary format, so they are not human-readable (let alone editable) in any easy way.

The scenario data should preferably be persisted using a better and less error-prone alternative: for example, an open-source Java library called **XStream** (available at http://xstream.codehaus.org) is able to serialize Java objects to and from a proprietary XML format that is also human-readable.

11.3 Putting it altogether

The object persistence features required by the scenario file are specified in the interface **ObjectSerializer**. Instances of this interface should be able to save objects to the specified target and load them from the specified source. The default implementation will use XStream, as discussed in the previous chapter.

The scenario file is accessible for the rest of the system via the class **ScenarioFile**. This class is basically a front-end to the Jar archive that contains the scenario data. It provides methods for saving and loading scenario data objects, such as notes and settings. Each of the methods for saving delegates the actual serialization of objects to ObjectSerializer, and stores the serialized object is as an entry in the Jar archive. The same applies to loading, vice versa (see figure 23). Each class wishing to save its state to the scenario file needs to implement the interface **Saveable**, which defines two methods: `getData()`, that allows the ScenarioFile to query the object for the data it wishes to save in the scenario file, and `setData(Object)`, which is used to pass loaded data back to an object instance.

The loading and saving of scenario data is coordinated by MainFrame, which keeps track of all the Saveable objects in the Animator.

12 Localization

In order to make the Animator universally available as a teaching tool, its user interface was designed to support many languages, i.e. to be easily localized.

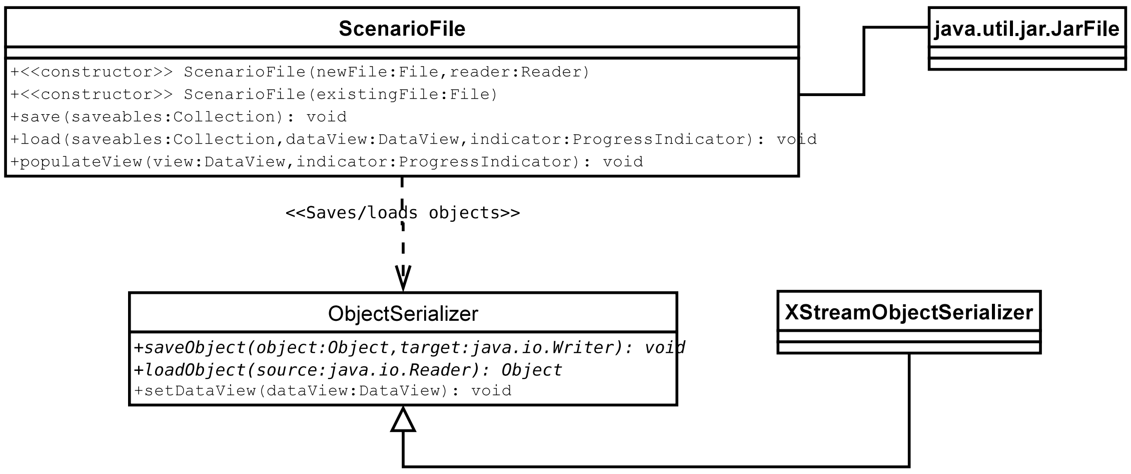

ScenarioFile
+<<constructor>> ScenarioFile(newFile:File,reader:Reader)
+<<constructor>> ScenarioFile(existingFile:File)
+save(saveables:Collection): void
+load(saveables:Collection,dataView:DataView,indicator:ProgressIndicator): void
+populateView(view:DataView,indicator:ProgressIndicator): void
java.util.jar.JarFile
<<Saves/loads objects>>
ObjectSerializer
+saveObject(object:Object,target:java.io.Writer): void
+loadObject(source:java.io.Reader): Object
+setDataView(dataView:DataView): void
XStreamObjectSerializer

Figure 23: Scenario file class diagram

12.1 The Localization class

The localization functionality of the Animator is centralized in one central class, **Localization**. At startup, the Localization class loads the property file locales.properties from the classpath. This file contains a list of all the available translations for the Animator. For each available language, the localized strings are stored in a file called **dacopanxx.properties**, where **xx** is a two-digit language code recognized by the java.util.Locale class. As English is the default language of the Animator, strings localized to English are stored in the file dacopan.properties. The standard Java class ResourceBundle takes care of resolving take localized strings from the property files.

The Localization class provides the method **getString(String key)** for retrieving the localized strings from the property files. A localized string is not restricted to static text only. It can contain placeholders for variables in the format supported by the class java.text.MessageFormat. To populate a string with variables, the method **getString(String key, Object[] params)** can be used.

For the purposes of listing the available languages in the user interface, the method **getAvailableLanguages()** can be used. To change the active language, use the method **setCurrentLanguage(Localization.Language)**. Localization.Language is a simple inner class that wraps the name of a language and the Locale it represents. The selection of a language is persistent: it is saved using the Java preferences framework (see package java.util.prefs).