Christian-Albrechts-Universität zu Kiel

Master Thesis

# View Management for Graphical Models

Martin Müller

29th December 2010

Department of Computer Science
Real-Time and Embedded Systems Group

Prof. Dr. Reinhard von Hanxleden

Advised by:
Dipl.-Inf. Hauke Fuhrmann

## Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kiel,

## Abstract

Creating and maintaining graphical models is a complex task. A lot of development time can easily be consumed by changing the visual representation of the model to match the current requirements. In this master thesis I introduce the KIELER View Management (KiVi) framework. It supports the dynamic generation and modification of views on a graphical diagram and comes with a wide variety of default applications. Additionally external developers can easily base their graphical modification tasks on the view management framework. The project connects existing KIELER technologies such as automatic layout and structure-based editing with new approaches for editing, viewing, or simulating graphical models.

# Contents

# List of Figures

# 1 Introduction

The development and specification of both software and hardware has evolved throughout the past decades. Crucially, the complexity of these systems has grown exponentially. Taking the Linux Kernel as an example, even what is basically the same set of features grows bigger and bigger over time. During the one-year development period from version 2.6.31 to version 2.6.36 the lines of source code have grown by 12%[1]. Keeping track of this growth using traditional methods is next to impossible.

Similar trends exist in virtually every development project. In order to cope with this growing complexity, graphical model-based design has emerged. During the inception phase of a new project, UML diagrams [6] help the various team members formulate their ideas. A shared understanding of the semantics lets both adept developers and less tech-savvy designers or even customers communicate on common ground. Once they have agreed upon a final design, model-driven development tools can generate basic code to launch the actual implementation effort from. Creating UML diagrams can be a tedious undertaking. Adding a new common super class for a set of classes requires a larger number of mouse interactions. Any change to the diagram may result in time-consuming rearrangement of the surrounding elements. Maintaining the mental map when working with large models is very hard. Every reasonable graphical model-based design environment must address and solve all these problems.

Parts of the final system may have been developed using SyncCharts [2]. Together with an execution engine such as Ptolemy [4, 9] these models can be simulated in order to test their behavior before building the actual system. In order for a human to follow such a simulation, the employed environment needs to provide various visual cues. The focus of the user must be guided towards the most relevant sections of the diagram, traditional zooming can not cope with the desire to see both minute details for important parts and at the same time to maintain a rough overview of the entire execution. These requirements demand an entirely new approach to managing the way a model is displayed.

Changing the look of a diagram by hand is very tedious and time-consuming, even when no modification is done to the semantic meaning of the underlying system. If there are multiple different aspects to depict with a view on the model each, the developer typically is forced to pick exactly one of these when modelling by hand. This can make the design much harder to understand when looking for a facet that is not clearly shown

[1]`http://www.h-online.com/open/features/What-s-new-in-Linux-2-6-36-1103009.html?page=6`

by the chosen single diagram view. The original developers may get along fine with this situation. Their customers or new team members may not. A dynamic means of creating different views for one model addresses this problem. Instead of sticking to one way of looking at the diagram, a view management framework allows for the quick creation of various shapes and appearances. This master thesis provides the core components for such a framework called the KIELER View Management (KiVi) along with several example feature implementations.

## 1.1 Related Work

There have been multiple previous papers working on the aspects of dynamic view management. I specifically want to highlight two theses that have introduced several approaches used throughout the view management design and implementation.

Jacobs [8] has compiled an extensive overview of different visualization concepts in his student research project. He concentrates on data flow languages, nevertheless many ideas are applicable to various types of diagrams as well. The most important aspect carried over from his thesis to the view management framework is the notion of semantic Focus & Context. In large diagrams traditional zooming fails to provide both overview and sufficient detail. In order to combat this issue Jacobs proposes to reduce the complexity of graphical elements that currently are not as important to the user, they are out of focus. This reduction could be achieved by locally different zoom levels reducing the size of less relevant entities, by hiding certain information such as long labels, or by collapsing higher levels of hierarchy as implemented by the SyncCharts execution visualization described in Section 5.3.6. Working together with automatic layout algorithms these procedures all aim to better utilize the available screen space.

Apart from the currently focused entities Jacobs also defines the associated context as objects directly related to those. Traditional zooming would show the focus in great detail but completely hide the context outside the editor's viewport. However, in order to fully comprehend these relevant entities the user needs to keep their siblings and other relationships in mind as well. Instead of cutting these off Focus & Context suggests to show them with a reduced level of detail. The KiVi implementation for SyncCharts realizes this by hiding the inner regions of a macro state if that state is part of the context.

In his diploma thesis Beckel [3] further elaborates view management concepts and provides a rudimentary realization of a core framework along with some example applications. Most importantly for KiVi he introduces the basic architecture of Triggers, Effects, and Combinations that has evolved into the four building blocks of KiVi. However, the accompanying implementation is considered experimental by the author. Sadly this assessment turned out to be very accurate, thus no part of that original source code could be recycled in the new view management framework.

## 1.2 Outline

This chapter has given a concise introduction into the subject matter of view management along with some of the problems this framework needs to solve. Chapter 2 summarizes the various employed technologies, especially from the Eclipse and KIELER universes. The design of the KiVi framework is described in Chapter 3. Its actual implementation along with details of the core components is illustrated in Chapter 4. The following Chapter 5 uses these core components for several concrete exemplary features employing the view management framework. Finally, Chapter 6 gives a summary of this master thesis and looks into potential future work.

# 2 Technology

This chapter introduces a selection of technologies utilized by the view management. It is built upon the open source Eclipse platform [11] that provides a rich set of frameworks for the graphical user interface, several pre-built editors, customizable preferences, and much much more. The view management focuses on the graphical modelling aspect of Eclipse, specifically the trifecta of the Eclipse Modeling Framework (EMF), the Graphical Editing Framework (GEF) and the Graphical Modeling Framework (GMF).

The KiVi framework belongs to the KIELER project[1] developed by the Real-Time and Embedded Systems Work Group within the Department of Computer Science at the Christian-Albrechts-University of Kiel. It is a research project aiming to enhance the graphical model-based design of complex systems, specifically with tools built upon the Eclipse Rich Client Platform. The focus is set on improving editing, viewing, and simulating large diagrams with features such as automatic layout algorithms, combined structure-based editing operations, or the integration of various simulation engines with a single execution manager.

## 2.1 Eclipse

Eclipse became popular as a powerful integrated development environment using object-oriented languages, especially for creating Java applications. For instance, its integrated Java application programming interface (API) content assist feature[2] has made looking for methods and classes virtually obsolete while developing new Java code. However, the scope of Eclipse is not limited to textual languages. There is a whole host of graphical editors for creating various model types, such as the UMLTools suite[3] for the Unified Modeling Language, or the Ecore editor[4] working with EMF to create a graphical model for data structures later represented by generated Java code. The view management focuses on the KIELER SyncCharts editor introduced in Section 2.2.1 as shown in Figure 2.1.

[1]`http://rtsys.informatik.uni-kiel.de/trac/kieler`

[2]`http://help.eclipse.org/helios/index.jsp?topic=/org.eclipse.platform.doc.isv/guide/editors_contentassist.htm`

[3]`http://www.eclipse.org/projects/project_summary.php?projectid=modeling.mdt.uml2-tools`

[4]`http://www.eclipse.org/projects/project_summary.php?projectid=modeling.emft.ecoretools`

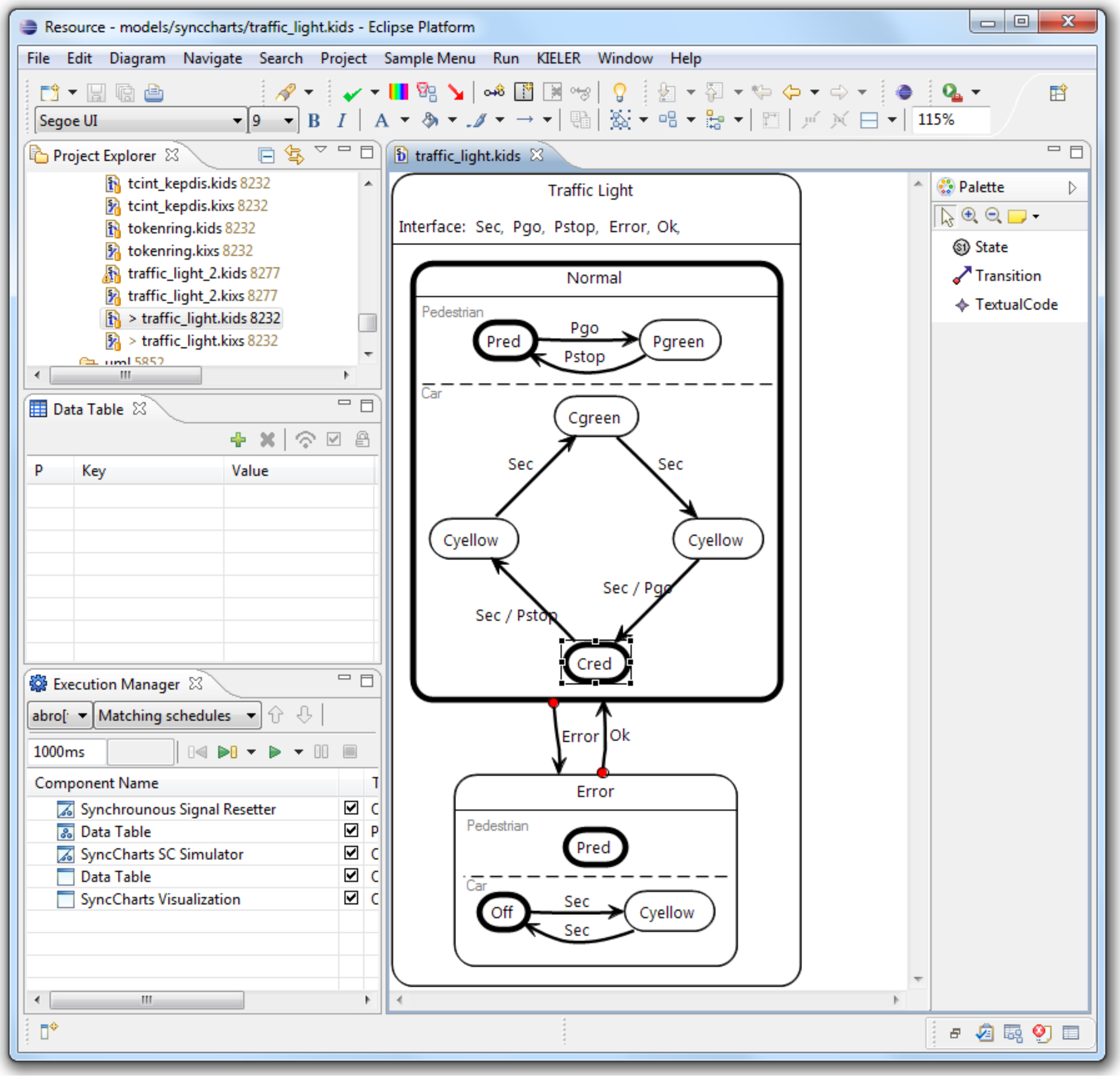


Figure 2.1: Eclipse running the KIELER SyncCharts editor.

### 2.1.1 Plug-ins

Every component of Eclipse is encapsulated in a plug-in, even the most elementary runtime core is one. Using this modular architecture, it is very easy to add or change some functionality without touching any foreign code. In order to facilitate a common interface between plug-ins, Eclipse uses the so-called extension point mechanism. Any plug-in can specify such an extension point as a socket for other plug-ins to connect to. For example, Eclipse provides an extension point for editors. Any new plug-in that contains an editor needs to connect to that extension point in order to be recognized as an editor. Within the extension point meta information it already specifies basic facts about what file types it can edit, or where the icon to display for this editor is located in the file system.

This meta information is rudimentarily important for actually making Eclipse work as a feasible application. A typical installation may contain hundreds, even thousands of plug-ins. Loading all of them on startup of the platform would not only take a very long time, it would exceed the memory capacity even of many modern computers. However, usually only a small percentage of plug-ins is used during a session. Eclipse exploits this fact by lazy loading of plug-ins. After reading and processing all plug-in meta data into a plug-in tree, only those plug-ins are loaded that will actually be used. When the user starts an editor for the first time, the editor will get loaded into memory in that moment. Using the plug-in tree, Eclipse can determine what plug-ins are necessary to run the editor without loading them exclusively for this task.

The lazy loading behavior is critically important for the view management framework. It will get used by many different editors and features, as a result the view management itself virtually always needs to reside in memory. Conversely, the plug-ins using the view management do not get loaded yet. The view management is necessary to run those editors, but those editors are not necessary to run the view management. Connecting various editors together in this way does not destroy the effect lazy loading has on the startup time of Eclipse and on its memory usage.

### 2.1.2 The Eclipse Modeling Framework (EMF)

The Eclipse Modeling Framework (EMF) [15] provides an integrated way to create Ecore meta models graphically within Eclipse. An Ecore meta model is closely related to a UML class diagram. It specifies classes with a set of attributes as well as the relations between different classes. Most conveniently EMF comes with a generator to transform the Ecore meta model into Java code. This generated implementation of the modeled data structures already comes with getters and setters for the specified attributes as well as code to serialize the objects into XML files and parse them back into objects. On top of the model representation, the EMF generator can also generate code for an entirely new editor specifically tailored towards the actual meta model. This takes the shape of a tree editor, effectively providing an easy-to-use way of modifying the tree representation of the data structures very similar to editing an XML file. The concrete generated implementation uses `EObjects` to denote the basic omnipresent superclass, much like the `Object` class in plain Java.

This capability makes EMF the ideal foundation for the generator-based creation of new editors within the Eclipse platform. These use the Ecore meta models built with EMF as the underlying definition of the files that can be produced with the new editors. In order to provide a graphical means of editing EMF teams up with GEF and GMF.

### 2.1.3 The Graphical Editing Framework (GEF)

EMF serves as the background meta model specification tool. The Graphical Editing Framework (GEF)[5] builds a visual representation for the domain-specific Ecore model. The corresponding components are called `EditParts`, typically there is at least one `EditPart` associated with each `EObject`. However, many complex entities may use more than one, for instance an arrow-shaped connection could consist of the actual arrow in one `EditPart` and several labels that each have their own `EditPart` as children of the previous one. The tree built this way resembles the structure of the EMF tree but is not identical. There may be missing entries denoting a model object that is not part of the current diagram, and there will be additional entries as mentioned above.

An `EditPart` implements editing facilities and serves as a link between the model and the concrete graphical representation. For that GEF relies on the Draw2D[6] framework, a lightweight rendering API on top of the Eclipse Standard Widget Toolkit (SWT) [10]. With its help `EditParts` can paint various `Figures` on their respective part of the diagram, such as rectangles, arrows, or simple text. This is where the view management regularly accesses and modifies the way a model object is displayed in the diagram.

### 2.1.4 The Graphical Modeling Framework (GMF)

Implementing a graphical editor manually using GEF is possible yet tedious, time-consuming, and error-prone. In the same spirit as creating the Java data structures from Ecore models with the EMF generator Eclipse provides the Graphical Modeling Framework (GMF). This is the missing link in the tool chain from model to a fully functional editor. The GMF generator takes the original Ecore model along with a set of user-defined specifications, such as the mapping of EMF entities to their respective graphical representation, as its input parameters and produces an entire editor. There still is more customization work to be done on in order to make the end result look and feel well-rounded, yet the basic editing features already are present. This generator-based approach massively simplifies and accelerates the use of GEF and Draw2D, effectively

[5] `http://www.eclipse.org/gef/gef_mvc/index.php`
[6] `http://www.eclipse.org/gef/draw2d/index.php`

allowing developers to create graphical editors for most domain-specific models they come across without having to manually program the largely identical functionality.

## 2.2 Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER)

The KIELER project enhances working with various graphical diagrams, especially facilitating the model-based design of new soft- and hardware. It evolved from the Kiel Integrated Environment for Layout (KIEL) [12], a standalone Java application that already incorporated features such as a means of calculating automatic layouts. There are several editors built into the KIELER universe, the view management focuses on the SyncCharts editor to showcase its functionality.

### 2.2.1 Thin KIELER SyncCharts Editor (ThinKCharts)

SyncCharts are an extended variant of Statecharts [7] designed to model reactive synchronous systems. The Thin KIELER SyncCharts Editor (ThinKCharts) is a GMF-generated editor with a rich set of customizations. Figure 2.2 shows the standard Hello World example featuring hierarchy, parallel regions, normal termination transitions, strong aborts, and many more aspects. SyncCharts serve as the prevailing test bed for the development of new features for the KIELER tool box. The hierarchical composition of SyncCharts provides an interesting use case for automatic layout algorithms. Additionally several simulation engines [9, 1] are capable of executing a SyncCharts model requiring a means of visualization.

### 2.2.2 KIELER Infrastructure for Meta Layout (KIML)

The KIELER Infrastructure for Meta Layout (KIML) [13] is one of the corner stones the KIELER project is built upon. It incorporates various layout algorithms such as the open source library GraphViz [5] into Eclipse diagram editors. Relieving the user of manually moving objects around the canvas is an enormous time saver and enables other plug-ins to add a rich variety of features. For instance, the visualization of SyncChart executions automatically collapses less relevant parts of the diagram. This would be unfeasible without a powerful layout framework.

### 2.2.3 KIELER Structure Based Editing (KSBasE)

Typically one modification to a graphical diagram actually entails multiple smaller operations such as adding a region to a SyncCharts state, inserting a new state to this

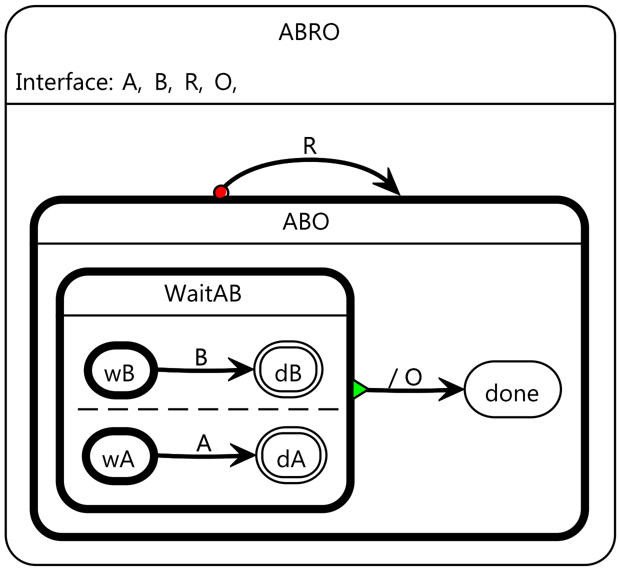


Figure 2.2: ABRO, the Hello World of synchronous programming in SyncCharts.

region, and flagging it as an initial state. These three steps together form one macro operation. The KIELER Structure Based Editing (KSBasE) [14] framework provides a means for model-to-model transformations. Its most prominent applications are such macro operations. For the example above the input model element would be the outer state. The transformation would perform the three steps all at once and swap in the output model, a hierarchical state containing one region with an initial state. On its own this would only modify the EMF objects but not the arrangement of the associated `EditParts`. This task is delegated to KIML by the view management.

### 2.2.4 KIELER Execution Manager (KIEM)

An essential aspect of model-based design is the ability to simulate a system without actually building it in reality. The KIELER Execution Manager (KIEM) [9] gives an interface for various execution engines such as Ptolemy or Synchronous C [1]. Additionally it allows for visualization components to receive data from the simulator, this is where KiVi connects with KIEM in order to translate that information into a colorful animated representation of the underlying sequence of events.

# 3 KIELER View Management (KiVi)

## 3.1 Overview

The *KIELER View Management* (KiVi) is a framework for the dynamic creation of different views for one diagram. There are several concerns that have to be addressed by the framework. First, the view management has to react to events or *Triggers* happening throughout the Eclipse platform. Second, every type of change to the current view must be encapsulated in an easily reusable generic *Effect*. Finally, these low-level building blocks are connected together on a higher level of abstraction by *Combinations*. The basic communications between these components is shown in Figure 3.1.

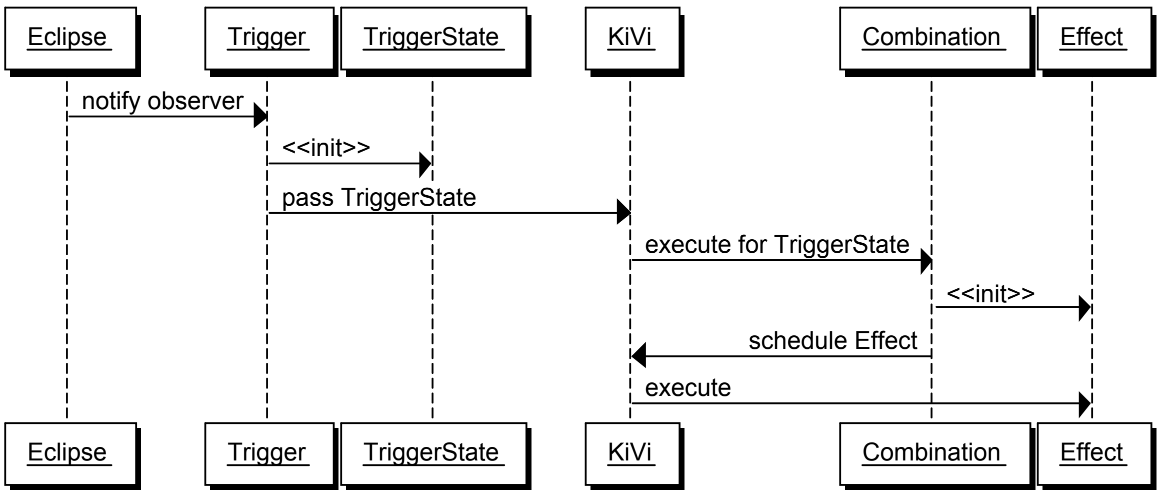


Figure 3.1: The simplified sequence of communication.

## 3.2 Triggers & TriggerStates

Any action taken by the view management is a reaction to some kind of event from the Eclipse platform. These events are observed by view management Triggers. However, working with events often is unnecessarily complicated. For example, during the simulation of a SyncChart an event may be "State S has become active". To maintain a complete overview over all currently active states each new event needs to be collated or *merged* to update the previous state with the changes contained in the new event.

To avoid redundant states and subsequent updates in every receiver of new events the view management provides *TriggerStates*. Every TriggerState belongs to exactly one Trigger and maintains the state associated with the events received by the Trigger by merging the previous state with the new event. Conversely, a single Trigger may produce multiple different types of TriggerStates. This reduces the demand for redundant Triggers listening to the same types of events.

Every Trigger registers itself with the entity it observes on demand. When there is no Combination waiting for an update to a TriggerState, the associated Trigger does not have to consume any resources. Analogously, when multiple Combinations require the same TriggerState updates, the corresponding Trigger only has to exist once. The view management will pass the new TriggerState to each Combination sequentially.

In order to implement a new Trigger the developer needs to have deep knowledge of the inner workings of Eclipse and the relevant plug-ins. Therefore the view management has to provide various Trigger and TriggerState implementations. This way the knowledge required to get started as a new developer is significantly reduced.

## 3.3 Effects

There are two kinds of Effects that contain every elementary view management action. Most ones only apply a transient change to the graphical representation of a diagram. Usually this modification is only temporary, for example to show an aspect of the diagram to the user for a few seconds. A common use case is highlighting the active states during the simulation of a SyncChart in a different color. These types of Effects are relatively light-weight and do not alter the underlying diagram or model files. The second kind is a group of more permanent Effects. These changes are meant to persist for a long time, typically until the next application of the Effect. The most prominent case is automatic layout, any newly calculated arrangement of objects must persist until the model is modified or a new layout is computed.

Many Effects need to be able to restore the previous state and negate any change they have applied to the diagram. For instance, an Effect that paints a diagram object in a different color must be capable of changing the object back to its previous color. There are two ways to achieve this ability. An Effect could read out some persistent property of the modified object and act accordingly, or it could memorize the previous value and later apply that stored value. The first method is a significantly cleaner approach. During intertwined execution of multiple Effects for the same diagram object the original value is unambiguously known. However, in some cases there is no mechanism of retrieving such properties. The second method can be used in every situation as long as the issues related to multiple executions of an Effect for the same object are handled or deemed irrelevant.

Similarly to Triggers, the implementation of new Effects requires intricate comprehension of the various Eclipse frameworks such as GEF and GMF. In order to allow more developers using the view management with less specific knowledge of the inner workings there should be a collection of commonly used Effects provided with the core view management framework.

## 3.4 Combinations

Triggers and Effects provide the low-level basis for the view management, much like the input and output of some system. The inner logic of that system corresponds to a Combination, connecting the input from Triggers with the output expressed as Effects. A Combination specifies a set of TriggerStates that are relevant for its execution. Every time one of these TriggerStates is updated by the firing of its associated Trigger the Combination is nudged into action. Along with this new TriggerState every other current instance of relevant TriggerStates is passed to the Combination. That way the entire picture of necessary information is presented, eliminating the requisite to maintain countless local variables within each Combination.

In contrast to Triggers and Effects, the end user or domain-specific application developer depends on being able to specify new Combinations to suit his use cases. For instance, his idea of how the visualization of a SyncChart simulation should look like may be entirely different from any visualization provided with the view management. In order to facilitate the later specification of Combinations, these have to operate on a much higher level of abstraction than Triggers or Effects. There must not be any intricate contact with Eclipse graphics internals such as GEF or Draw2D. A Combination developer instead only needs to focus on his own domain-specific models. Any diagram data received from Triggers through TriggerStates consists of EObjects representing the actual model objects underlying the diagram. Similarly, all Effects take EObjects as parameters or targets. Using this added level of abstraction a Combination does not tie itself to a specific graphics implementation, instead one could for example use the same Combination logic for both GMF and Graphiti[1] diagram editors.

## 3.5 View Management Controller

The trinity of Trigger/TriggerStates, Effects and Combinations is tied together by the view management controller. It monitors the user preferences and loads the appropriate Combinations on startup of the Eclipse platform. It takes care of instantiating and activating all Triggers registered by the Combinations. It keeps track of the set of Triggers a Combination is listening to. It receives new TriggerStates from the Triggers and distributes it to the corresponding Combinations. After the execution of a Combination it performs all Effects scheduled by that Combination. In other words, it handles all

[1] `http://www.eclipse.org/modeling/gmp/?project=graphiti`

aspects of the communication within the view management components as well as the input from and output to the Eclipse platform.

## 3.6 Concurrency

The view management framework must support concurrent executions in a robust way. This issue affects every component of the view management.

Triggers are called from any part of the Eclipse platform, this fact implies that new TriggerStates may be created in any order, interleaved with each other, or even truly in parallel on multiprocessor or multicore architectures. As a result every Trigger implementation needs to take care of concurrency problems in its public code on its own. For instance, a Trigger listening to changes to model files may get called by two unrelated changes in parallel. It has to be well-behaved in that scenario.

Similarly the view management controller itself must be able to handle a flood of incoming new TriggerStates. No TriggerState may be lost, even if many new instances of the same type of TriggerState already are in processing. On top of that, the order of TriggerStates must be maintained. In order to honor the observer pattern, Triggers need to handle the event from the Eclipse platform quickly. Typically any long computation would block parts of the UI or some other component vital to an interactive, responsive user experience. The solution to these two issues is the sequentialization of all incoming TriggerStates in a queue, maintaining their order. This queue is processed in a new Combination worker thread, enabling the instant return of all Triggers to let the Eclipse platform continue with whatever it was doing when the Trigger fired. Every incoming TriggerState still needs to be merged with the previous instance of that TriggerState to maintain a consistent state from potentially partial events. This merging is best done by the Combinations worker thread just before the TriggerState is passed on to the relevant Combinations.

Using the sequentialization of TriggerStates by the view management controller, handling concurrency within Combinations becomes relatively simple. The Combination worker thread only executes at most one Combination at a time, as a result there is no requirement for synchronization and locks within their implementations. Therefore a developer later adding a new Combination has one problem fewer to take care of. The list of Effects requested by a Combination's execution is passed back as a return value. It gets added to the back of yet another queue by the view management controller. This Effects queue is processed by the Effects worker thread, ensuring the correct order of Effects.

Due to this thread an Effect itself can be certain that there is no other Effect executed at the same time, neither in parallel nor interleaved. This makes any internal synchronization unnecessary. However, most Effects perform some change to the graphical user interface. Within the Eclipse SWT framework, any access to the UI must be done by the special UI thread. Hence most Effects need to handle these operations accordingly.

To summarize, there are four threads working together within the view management framework. The first one fires a Trigger, this may be any thread throughout the Eclipse platform. The amount of processing done by this thread is minimal, all that happens is the creation of a new TriggerState and its addition to the back of the CombinationsWorker queue. Now this arbitrary foreign thread can return and continue, for instance ensuring the responsiveness of Eclipse if a Trigger was fired from within the SWT user interface thread. Straight away the CombinationsWorker takes over and executes the list of Combinations that have registered themselves for this TriggerState. Each Combination returns a list of Effects that is enqueued with the EffectsWorker. That thread handles the actual execution of the Effects. However, some Effects modify the graphical user interface. These changes need to be performed from within the SWT user interface thread.

# 4 KiVi Implementation

## 4.1 Core Components

This section highlights implementation details of the core view management components. These reside in the Eclipse plug-in `de.cau.cs.kieler.core.kivi` and are part of the KIELER Core feature, resulting in two implications. First, there can be no Eclipse installation utilizing any KIELER functionality without also installing the view management core. Second, any KIELER developer can rely on having KiVi available for his own developments, and can therefore fully utilize the view management framework.

The four types of components that can be contributed to the view management by other plug-ins - Triggers, TriggerStates, Effects, and Combinations - are specified by an interface each. In order to simplify the task for future developers, common functionality has been implemented in four abstract classes. This is most relevant for Combinations because a lot of code-saving measures are present there. For instance, when using the abstract implementation the developer only needs to specify the TriggerStates required by this Combination once. Any related tasks such as providing a processable list of them for the view management controller happens behind the curtains using the Java reflection API.

### 4.1.1 Extension Points

In order to let foreign plug-ins contribute features towards the view management, KiVi has to provide extension points with the standard Eclipse mechanism. Only one extension point is necessary, it registers every Combination that shall be handled by the view management under the id `de.cau.cs.kieler.core.kivi.combinations`. In this set of meta data the developer needs to specify the full Java path to the Combination class along with a user-readable name that is displayed in the KiVi preference pages. Optionally there can be a description to provide more detailed information about the Combination. Additionally the developer can specify whether the Combination shall be activated on startup or not. If this value is not present then the Combination will be inactive. Any user preference takes precedence over these fall-back values, their main purpose is to ensure a working KIELER installation with well-defined settings for the first launch. This way a novice user can enjoy the view management features without going through the lengthy process of selecting those Combinations that virtually all users would require enabled anyway.

### 4.1.2 View Management Controller

The view management controller resides in the class conveniently called `KiVi`. It takes care of the communications from Triggers and TriggerStates to Combinations, as well as of the scheduling of new Effects. During startup of the Eclipse platform this class is instantiated as part of the singleton pattern. This initialization procedure reads out the information from the Combinations extension point. From the user preferences and the fall-back extension point information it determines what Combination needs to be instantiated and activated. Additionally, it collects the set of Triggers and TriggerStates that those Combinations require and initializes them. If there are multiple Combinations requesting the same Trigger then only one instance is created. An incoming TriggerState from this Trigger will be distributed to all Combinations that ask for it. This $n : m$ relation between TriggerStates and Combinations is stored within the KiVi class for quick access.

#### EffectsWorker

This `EffectsWorker` class in the `internal` subpackage takes care of executing every view management Effect sequentially. Some may take a while to run, using a separate Thread avoids blocking the UI with a slower Effect. The aforementioned startup procedure also launches this Thread, when activated it immediately goes to sleep waiting for new Effects. These are stored in a queue to maintain the execution order. As long as this queue is not empty the worker will dequeue an Effect and execute it. As mentioned in Section 3.3, Effects can be merged with each other. This reduces the overall processing load, and avoids ugly flickering when some Effect is quickly undone after execution. This merging happens when an Effect is enqueued with the EffectsWorker by the view management controller. The method iterates over the queue and merges the new Effect with any existing Effect if possible. The result of a merge is another Effect, this one will be used in place of the newly enqueued Effect. The existing Effect that was used for merging is discarded. If there still are any more existing Effects left, the merged Effect is further merged with those where applicable. The effectiveness of this mechanism is best observed with the `LayoutEffect` described in Section 5.2.2, in short it performs the KIML automatic layout. If for some reason there are multiple Effects queued that all want to compute a new layout for the same diagram, there is no requirement to run all of these Effects - one execution will suffice. Using the merging mechanism, only the latest LayoutEffect will be executed, any earlier queued Effect calling the automatic layout for the same diagram is merged away.

Any Effect executed by the EffectsWorker needs to be treated as foreign and potentially unsafe code. If there are any uncaught exceptions the thread may not terminate. As a result any exception thrown by the execution of an Effect is caught by the EffectsWorker and its contents are reported to the user. This way poor programming in some plug-in using the view management does not cripple the entire system until it is restarted.

Some Combinations or other Eclipse or KIELER components may want to react to the execution of an Effect. For instance there is an Effect that collapses and expands complex graphical nodes. This action invalidates the existing layout, either freeing some of its space or taking up more space than before. As a result, a Combination listening to this Effect being executed can automatically schedule a new layout computation to restore an aesthetically laid out diagram. To facilitate this mechanism the EffectsWorker allows for listeners to register themselves for notifications. These are created when an Effect has finished its execution and contain a reference to that Effect. This observer feature is utilized by the `EffectsTrigger` introduced in Section 5.1.3 that fires when an Effect has been executed.

### CombinationsWorker

Similarly to the EffectsWorker described above, there is a CombinationsWorker that handles the execution of all Combinations. This thread is significantly less complicated because there is no merging mechanism to be done, and there is no need for any observer to listen to Combinations being run. Its only job is to accept new TriggerStates by the view management controller, to store them in an internal queue, and to sequentially execute the corresponding Combinations.

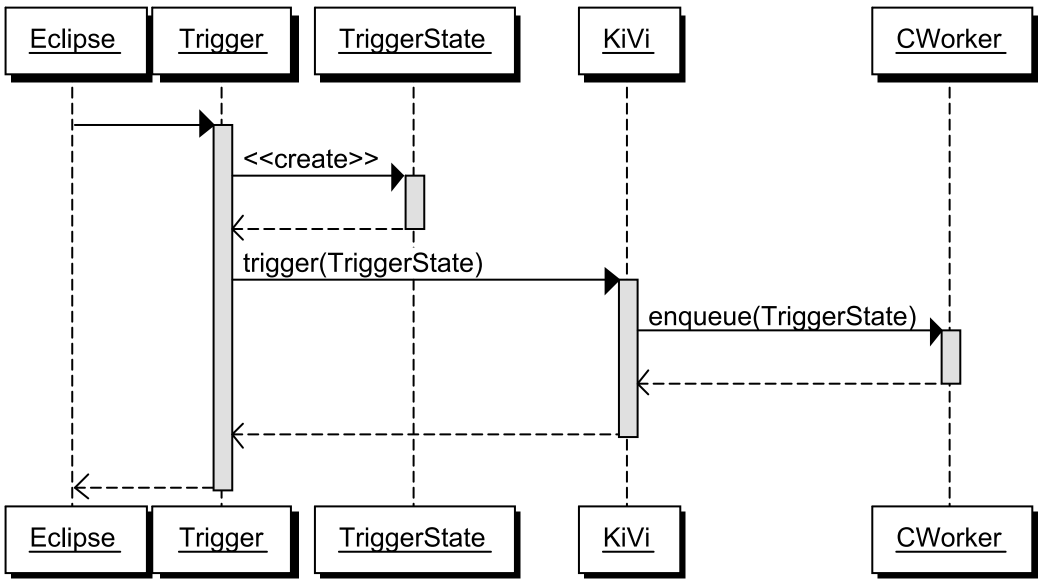


Figure 4.1: Sequence diagram showing the Trigger components.

This execution logic is contained within the view management controller but called from within the CombinationsWorker thread, again to not block the observing Triggers for longer than absolutely necessary. The distribution of a TriggerState object to its Combinations works as follows. First, the TriggerState needs to be merged with the previous instance of the same type. To recall from Section 3.2, this merging of TriggerStates

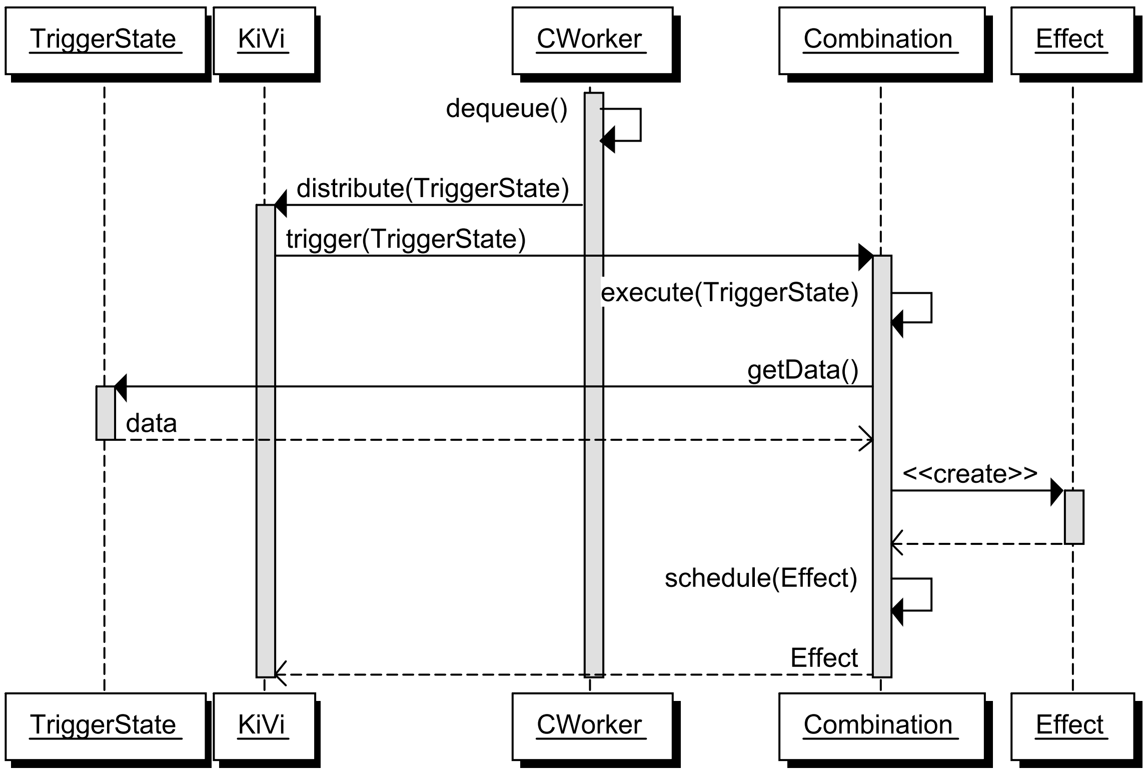


Figure 4.2: Sequence diagram showing the Combination components.

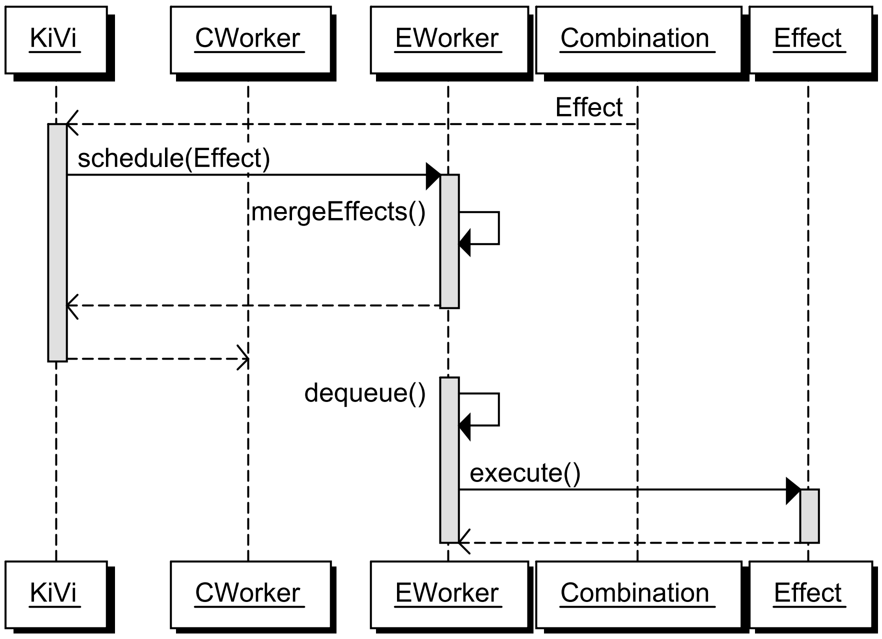


Figure 4.3: Sequence diagram showing the Effect components.

allows for delta-style events to be translated into the full representation of the corresponding state making their handling by Combinations significantly easier. Now the list of Combinations that have registered for this type of TriggerState is compiled from the internal storage of the controller and iterated over, calling the `trigger` method of each Combination sequentially. This method returns a list of Effects that are scheduled by the Combination. That list gets passed to the EffectsWorker for execution.

Figures 4.1, 4.2 and 4.3 show comprehensive sequence diagrams for each view management component ranging from the first notification by an observed part of Eclipse to the execution of a resulting Effect.

### 4.1.3 Trigger

Every view management Trigger needs to implement the `ITrigger` interface. For the convenience of future developers there is the `AbstractTrigger` class taking care of the more mundane tasks. The interface methods are as follows:

- `void trigger(ITriggerState triggerState)`
  The `trigger` method is a shortcut for passing the TriggerState parameter to the view management controller, it is therefore included in the abstract Trigger class. When the observer part of a Trigger receives an event it calls its own `trigger` method. This way the developer does not have to worry about how the TriggerState actually gets passed to the view management controller, reducing the amount of code that would have to be changed if that mechanism were to change in a future version.
- `void setActive(boolean a)`
  The view management calls this method to set the Trigger as either active or inactive. The abstract implementation maintains a local variable for this state and calls the `register` or `unregister` methods accordingly.
- `boolean isActive()`
  This is a simple getter for the active state of the Trigger, usually there is no need for any additional work beyond the abstract method.
- `void register()`
  `Register` has to be implemented by the Trigger developer. It is called automatically by the abstract Trigger class whenever the Trigger is activated, and shall take care of registering itself with the observed entity. For instance, a Trigger observing changes to the active selection would need to get added as a new selection listener to the Eclipse selection service.
- `void unregister()`
  Analogously to the `register` method, this handles any action taken when the Trigger is deactivated, such as removing itself from any observed objects or freeing any used resources.

Additionally to these methods the developer still has to program the actual Trigger logic. Typically this involves implementing a bespoke interface provided by the observed entity along with registering the class as a listener with the appropriate plug-in through an extension point.

### 4.1.4 TriggerStates

The interface specifying all common public methods of TriggerStates is called `ITrigger-State`. Again, several typical duties have already been implemented in the `Abstract-Trigger` class.

- `void merge(ITriggerState previous)`
  The `merge` method takes care of maintaining the actual state of an observed value from delta-style events. It is called on every new TriggerState instance with the previous object of the same type as the parameter. This way the latest update can be applied to the old state. By default this is an empty method because there is no means of merging applicable to all TriggerStates. Any concrete subclass that works with incremental update events needs to insert its own merge code.
- `void finish()`
  Some Trigger information is only valid for the originally launched batch of Combination executions. The most common example is a button in the Eclipse tool bar menu. There would be an event denoting the pressing of said button, however there would not be any event stating that the button is not pushed in anymore. If the associated TriggerState were to maintain the information from the initial event, every subsequent Combination execution would falsely assume that the button has just been pressed. To allow for the correct behavior, any TriggerState working with such an event type can specify how to invalidate the information in the `finish` method. It gets called by the view management controller after the list of relevant Combinations has received the TriggerState for the first time. Again, the abstract implementation is an empty method to clean up subclasses that do not demand this functionality.
- `Class<?  extends ITrigger> getTriggerClass()`
  Every TriggerState belongs to exactly one Trigger, while a Trigger may produce several different types of TriggerStates. In order to tell the view management which Trigger must be activated when a Combination registers itself for this TriggerState, the `getTriggerClass` method returns the class of the associated Trigger. This is the only location where the developer of a Trigger-TriggerState group has to specify their relationship, making any subsequent changes to the design as simple as possible.
- `Class<?> getKeyClass()`
  The view management controller keeps exactly one instance of every class of TriggerState. However, there may be some that use a Java generics type parameter.

For instance, the TriggerState denoting that an Effect has just finished its execution uses the class of its Effect to distinguish itself from TriggerStates for different Effects. As a consequence of this fact the class of the TriggerState can not be used as an identifier for determining what old TriggerState needs to be merged with an incoming one. In order to allow for these versatile TriggerStates, the `getKeyClass` method provides an alternative key to the view management controller. By default this maps to the `getClass` method and has to be overwritten if the concrete implementation requires a more dynamic key.

- `long getSequenceNumber()`
  A Combination may register itself with an arbitrary number of TriggerStates. Upon execution it receives the most recent instance of each type. There may be Combinations that would like to know the order in which the TriggerStates occurred, or what TriggerState is responsible for the execution by having fired the latest. The sequence number mechanism implements a crude but robust way of providing this information to Combinations. Before a new TriggerState is passed on the abstract implementation sets its sequence number using a globally unique counter, ensuring that the order of TriggerStates can always be determined when necessary.

For the convenient usage of TriggerStates by Combinations, and in order to allow Eclipse's content assist feature to work properly, every TriggerState should provide direct getter methods for any data it holds. These should return sane default values in case a Combination execution requiring TriggerStates A and B is started by A before B was fired once since startup. In that case the Combination will receive the result of the default constructor. Performing `null`-checks and the like in every Combination would not only clutter the code unnecessarily, it would also introduce countless programming error generators. If there is TriggerState information contained in a list, that getter may never return `null`. Instead it should create an empty list on demand and pass this back to the caller.

### 4.1.5 Effects

Following the same design philosophy as with Triggers, Effects come with the interface `IEffect` that needs to be implemented by every view management Effect as well as with an abstract implementation to make the developer's life a bit easier.

- `void execute()`
  The `execute` method contains the actual logic that performs the changes intended by the Effect. It gets called from within the EffectsWorker thread, as a result the developer does not need to be concerned with synchronization issues between multiple Effect instances. However, most Effects modify the user interface in some way. Any change to the user interface in Eclipse needs to be run by the bespoke SWT user interface thread. Hence special care needs to be taken in those cases, typically by specifying an anonymous `Runnable` that subsequently is enqueued with the user interface thread.

- `void undo()`
  Some Effects modify an aspect of a diagram temporarily, such as painting a node in a different color. That change needs to be undone at a later point in time. The abstract implementation of the `undo` method is empty because some Effects can not be undone.
- `boolean isMergeable()`
  This is a convenience method to speed up the merging mechanism that removes redundant Effects from the EffectsWorker queue. If an Effect can not be merged, there is no need to iterate over the list of existing Effects. Since not all Effects are mergeable, the `isMergeable` method returns `false` unless overwritten.
- `IEffect merge(IEffect otherEffect)`
  The actual merging of Effects happens in this method. It gets another Effect passed as a parameter and determines whether this instance can be merged with the other Effect, and if it can be merged it returns the new Effect. If the two Effects can not be merged, for instance if they operate on different diagrams, then the method returns `null` to denote this failure. This is the default abstract implementation. When successful, both old Effects are discarded and the new Effect is used in place of the called instance. Additionally, the merged Effect will receive more `merge` calls in case the remainder of the queue still contains some redundant Effects.

### 4.1.6 Combinations

Combinations tie Triggers and Effects together, converting the input of a set of TriggerStates into a list of user interface changes. Once again there is the interface `ICombination` specifying the obligatory public methods and an abstract implementation that provides the foundation for all concrete classes. However, while the abstract Trigger and Effect classes only took care of mundane tasks, the `AbstractCombination` actually contains a significant amount of "black magic". This background functionality transforms the way to specify a new Combination.

Most importantly, the `AbstractCombination` maintains a cache of the most currently executed Effects. When the next run of this Combination has determined a new set of Effects, the previous ones are scheduled for undoing. The new set of Effects might contain an Effect that would re-do such an undo. In order to avoid unnecessary processing and ugly flickering graphics, these Effects are merged with each other. Effectively an undo is consumed and discarded by an Effect that would recreate it. Another benefit of this feature is the lack of any housekeeping in the concrete Combination. There is no need to kept track of what Effects are active, what Effects need to be undone, what Effects need to be changed. The current execution simply specifies the state of Effects that shall be achieved, and the merging mechanism takes care of only applying the differences.

Similarly to TriggerStates specifying their associated Trigger, a Combination only needs to specify its TriggerStates in a single location - by declaring an `execute` method that takes the TriggerStates as concrete parameters. The abstract implementation will not only extract a processable list from these method parameters through the Java reflection API, it will also make sure every parameter is filled with the most recent instance of the appropriate TriggerState upon each execution.

The amount of abstract code for Combinations becomes clearly visible when comparing the length of the previous components. Adding those three together still yields fewer lines of code than the `AbstractCombination` alone. Due to this great batch of preprogrammed features implementing a new Combination becomes a relatively quick and easy task. The following methods are part of the `ICombination` interface.

- `void undo()`
  `Undo` is called by the view management when a Combination is deactivated, typically from the user changing the KiVi preferences. If there are any previous Effects still active the abstract implementation will schedule them to be undone by the view management controller.
- `List<IEffect> trigger(ITriggerState triggerState)`
  This is the entry point invoked by the view management controller for a new TriggerState after it has been merged with its predecessor. The first part of "black magic" is contained in the abstract implementation of this method. Initially it determines the list of TriggerStates necessary from the parameters of the `execute` method. Then it retrieves all the most recent instances of these types and calls `execute` passing this list. In return it receives the Effects that shall be executed by this Combination run. There still is a cached list of the Effects from the previous execution. These are marked to be undone, and merged with the new Effect list. This removes any redundancy resulting from the approach of specifying the state of Effects in the concrete `execute` method rather than only giving changes that need to be applied. The remaining merged Effects are cached for the next run and passed back to the view management controller for their actual execution. A Combination developer can choose not to use the abstract implementation and program a differing trigger mechanism as long as it still complies with the interface declaration.
- `Class<? extends ITriggerState>[] getTriggerStates()`
  As described above, the Combination developer only specifies the list of TriggerStates required for execution once - as concrete parameters to the `execute` method. In order to provide this list in processable form the abstract implementation converts these parameters using the Java reflections API. A fully custom-built Combination not extending `AbstractCombination` would need to deliver its own definition of the TriggerStates list in this method.
- `void setActive(boolean active)`
  Calling this method tells the Combination to activate or deactivate itself. Since this might be run from outside of KiVi, the Combination still needs to notify the view management of its new state. This functionality already is provided by the

abstract implementation.

- `boolean isActive()`
  Similarly to Triggers, this is a simple getter to determine whether this Combination is active or not.

On top of these interface methods there are several important ones used by the mechanisms implemented in the `AbstractCombination`. Any Combination developer needs to be well aware of these in order to fully profit from these benefits.

- `public void execute(TriggerState1, TriggerState2, ...)`
  This method is not part of any interface or abstract superclass, hence the programmer can not rely on the standard aids such as Eclipse's content assist feature. Because the number and types of parameters that may be specified for this method is variable throughout different classes, but constant for any single class there can be no common declaration with any Java mechanism. The only way closely resembling this flexibility would be the variable length argument lists denoted by three dots after the type declaration such as in `double average(double...numbers)`. However, this would only allow for a list of parameters of the same class or any of its subclasses. Not only would this mechanism destroy the reflection features mentioned above that remove the need for redundant specification of TriggerStates relevant to a Combination, it would also disable any help from the content assist. Declaring the concrete class for a TriggerState lets the developer view the available getters and access the information he needs quickly.

  Apart from implicitly declaring the list of TriggerState types that are required for the execution of this Combination, the `execute` method contains the actual logic to be run. A typical implementation would first access data from the TriggerStates, then perform some calculation on said data, and finally schedule Effects using the methods introduced below.

- `protected void schedule(final IEffect effect)`
  In order to enqueue an Effect with the EffectsWorker one could call the appropriate method of the view management controller. Doing that would require additional knowledge of the inner workings of KiVi though. Instead, there is the `schedule` method implemented by the abstract Combination class. Now a developer can simply look for methods his own Combination already provides and does not need to read as much documentation to achieve the same end result. Additionally, changing the internal design of the view management controller becomes much easier with this kind of encapsulation. Any modification would not have to be applied to every Combination in potentially countless distributed projects, there only needs to be one change in the `AbstractCombination`.

- `protected void doNothing()`
  To recap, the abstract implementation of the `trigger` method would automatically undo the list of Effects that were scheduled during the previous Combination run, and it would then enqueue the new list of Effects both with the EffectsWorker

and its internal cache. However, some Combinations may not want to perform any changes to their Effects state after an execution. For instance, it might check whether some condition is true and only become active in that case. Quietly returning the `execute` method would implicitly request an undo on the previous Effects. Calling `doNothing` first modifies the behavior of the abstract implementation in such a way that the Effects remain unaltered, as if this Combination execution never happened.

- `protected void dontUndo()`
  This method serves a similar purpose. Calling it before letting the `execute` method return disables the automatic undo function in the abstract Combination implementation during this execution. Using this feature the previous batch of Effects will persist indefinitely instead of getting undone automatically.
- `public static CombinationParameter[] getParameters()`
  Declaring this method is optional, its presence is checked with the Java reflections API. By returning an array of `CombinationParameter`s a Combination can dynamically add field editors to the KiVi preference pages. These settings are necessary to allow flexible Combinations instead of creating a new one for every minute difference required in some situation.

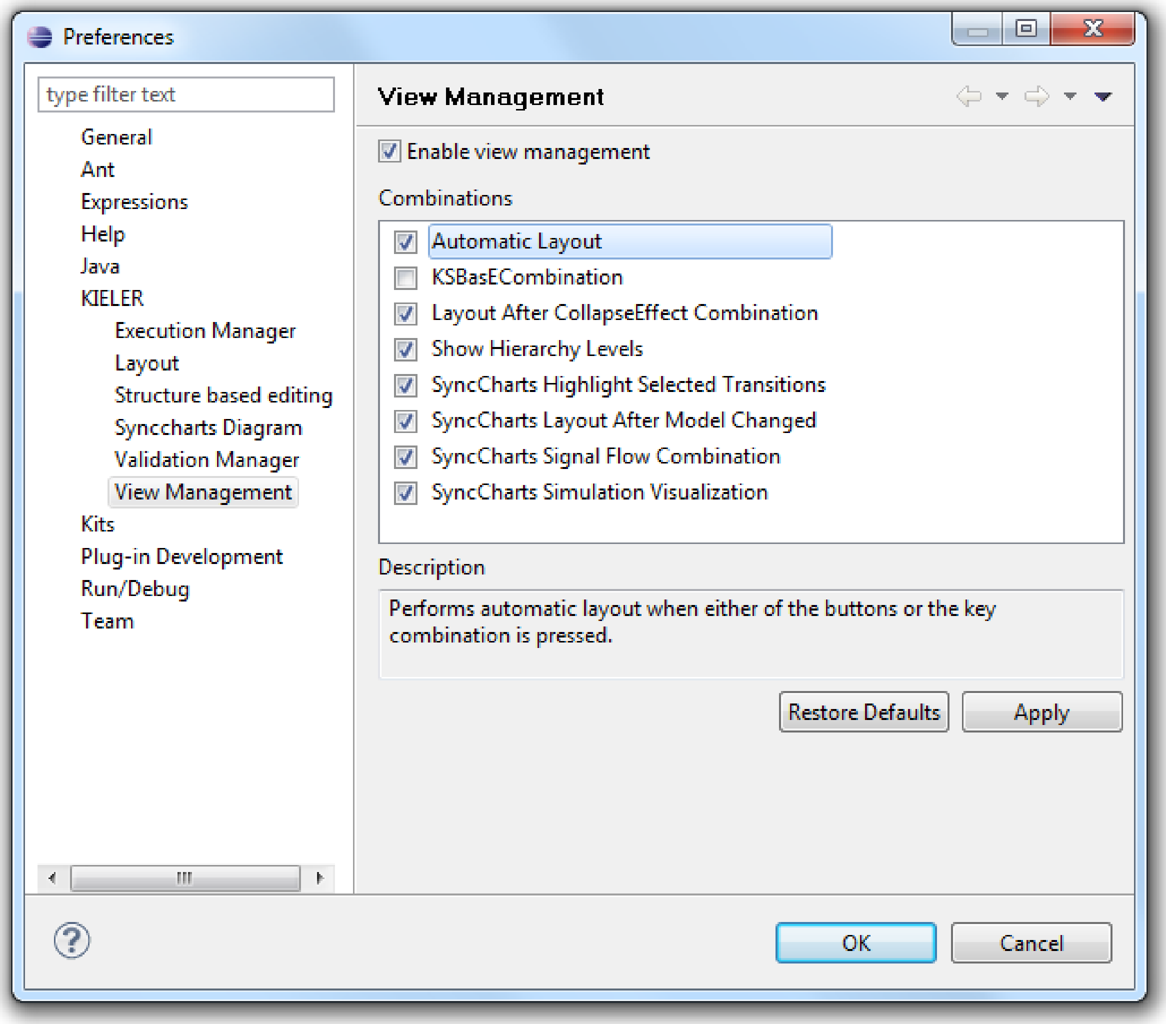
The aforementioned `CombinationParameter` class serves as a communication standard between Combinations and the view management preferences. It only serves as a data storage, there is no program logic contained. Specifically, this class holds the identifying key of the parameter and the preference store used, a readable name and description to display to the user in the preference page, the default value, and the type of parameter described.

## 4.2 Preferences

The view management framework comes with two dynamically generated preference pages. As opposed to purely static ones, various components can contribute content to these pages without actually modifying the Java source code of the preference pages or directly using any graphical user interface elements such as buttons or field editors. Instead, there are several well-defined ways of adding a new option to the existing pages.

The first page displayed in Figure 4.4 handles global settings. It comes with a check box to enable or disable the entire view management framework as well as with a check box list of available Combinations. These are displayed using the readable name specified in the Combinations extension point. On top of that the user can select a Combination from the list and view its extended description. Typically there is very little need to modify these boolean preferences because most Combinations already come with sane default values.

After those global settings the second page takes care of local Combinations preferences



View Management

[✓] Enable view management

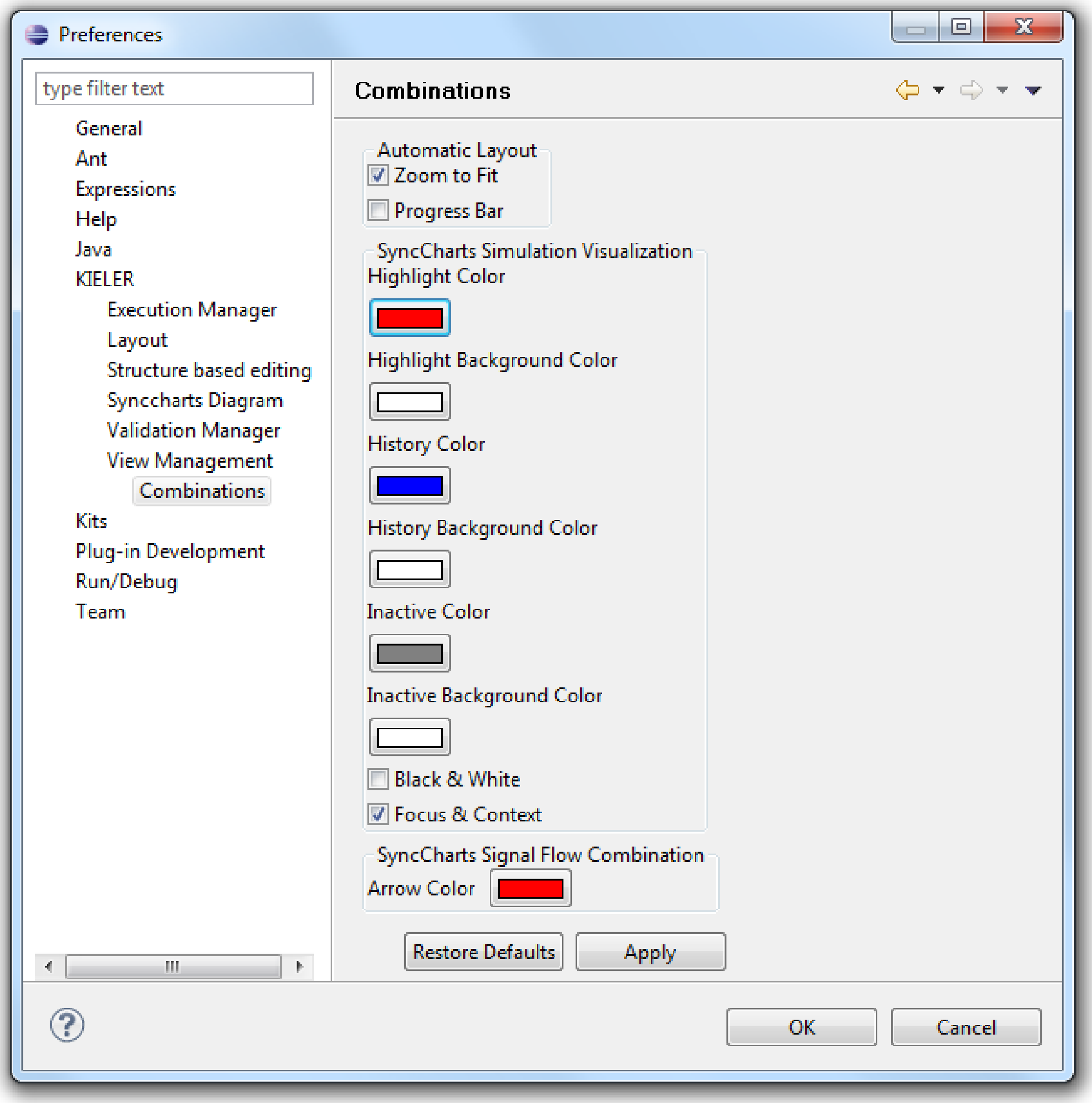
Combinations

- [✓] Automatic Layout
- [ ] KSBasECombination
- [✓] Layout After CollapseEffect Combination
- [✓] Show Hierarchy Levels
- [✓] SyncCharts Highlight Selected Transitions
- [✓] SyncCharts Layout After Model Changed
- [✓] SyncCharts Signal Flow Combination
- [✓] SyncCharts Simulation Visualization

Description

Performs automatic layout when either of the buttons or the key combination is pressed.

Figure 4.4: The global view management preference page



Preferences

type filter text

- General
- Ant
- Expressions
- Help
- Java
- KIELER
  - Execution Manager
  - Layout
  - Structure based editing
  - Syncharts Diagram
  - Validation Manager
  - View Management
    - Combinations
- Kits
- Plug-in Development
- Run/Debug
- Team

Combinations

Automatic Layout
- [✓] Zoom to Fit
- [ ] Progress Bar

SyncCharts Simulation Visualization
- Highlight Color 🟥
- Highlight Background Color ⬜
- History Color 🟦
- History Background Color ⬜
- Inactive Color ⬛
- Inactive Background Color ⬜
- [ ] Black & White
- [✓] Focus & Context

SyncCharts Signal Flow Combination
- Arrow Color 🟥

Figure 4.5: The local Combinations preference page

as shown in Figure 4.5. Each Combination can specify an array of `CombinationParameter` instances that declare the globally unique key string within the preferences, the preference store where the variable shall be saved in, a readable name and description, and the type of Variable along with its default value. There are pre-defined parameter types as well as the appropriate field editor user interface elements for strings, integers, floats, doubles, booleans, and 24-bit RGB color values.

# 5 Implemented Features

The previous chapter has described the core components of the KIELER View Management framework. It provided the backbone, but does not have any visual impact on its own. That is achieved by various exemplary feature implementations for commonly used Triggers with their associated TriggerStates and Effects as well as a couple of Combinations putting them to work. While the core view management components can be used for any graphical framework, these concrete classes are tailored to work with GMF editors.

## 5.1 Triggers & TriggerStates

### 5.1.1 SelectionTrigger

The `SelectionTrigger` listens for changes to the current selection within any graphical Eclipse editor. It resides in the actual KiVi package, allowing every plug-in to use this Trigger freely. In the `register` method defined by the `ITrigger` interface the `SelectionTrigger` adds itself as a new listener to the Eclipse selection service. This uses the `ISelectionListener` interface declaring a single `selectionChanged` method, receiving the editor where the selection occurred and the actual list of selected `EditParts` as parameters. Using this information the Trigger creates a new `SelectionState` instance. On top of the standard methods that provides two getters, one for the GMF `DiagramEditor` and one for the list of EMF `EObjects` that were selected. This adheres to the convention of using the domain-specific model objects rather than the generic diagram objects in communications between the various view management components. Every time the user changes the selection, for instance by clicking on a graphical node, this Trigger is fired updating the TriggerState. As a result, the most recent `SelectionState` instance always contains the current list of selected model objects. Knowing this a Combination developer easily retrieve that information by adding the `SelectionState` to the parameter list of the `execute` method. Storing that list within the Combination on every firing of the `SelectionTrigger` is unnecessary because the most recent TriggerState is available in every execution, providing quick access without additional effort.

### 5.1.2 ButtonTrigger

Usually any Eclipse project that wants to place a button anywhere in the menus or toolbars needs to conform to a large number of Eclipse standards, interfaces and extension points. Hence the amount of knowledge necessary to get a simple input from the user interface is considerable. The view management `ButtonTrigger` reduces this effort significantly, it is part of the core package and hence is available to all users of the view management. Only the actual definition of the button itself with its name, icon, or hover tooltip must be done through the appropriate Eclipse menu contributions extension point. Everything else is taken care of by this Trigger. It comes with a handler that listens to every view management button registered in this way. When executed, this handler interprets the event received from the Eclipse user interface and creates a new `ButtonState` instance. This TriggerState contains every bit of information collectable, the active editor, a string to identify the button pressed, a map of any potential parameters of the button, and a flag to determine whether a toggle switch has been flicked into the on or off position. Now an arbitrary number of Combinations can easily receive these button events without first digging deep into the inner workings of the Eclipse menu structures in the spirit of simplifying the development of new Combinations as far as possible.

### 5.1.3 EffectTrigger

Some Combinations may want to react to the execution of an Effect. For instance, the `CompartmentCollapseExpandEffect` described in Section 5.2.4 invalidates the diagram layout by either collapsing or expanding a compartment, effectively changing its size. Instead of hard-wiring the computation of a new automatic layout at the end of the Effect, listening to the collapse Effect getting executed in a new Combination retains more flexibility. The `EffectTrigger` provided along with the view management itself allows for this mechanism to work by registering itself as a listener with the `EffectsWorker`, and receiving a notification every time an Effect has finished executing. That information is translated into a new `EffectTriggerState` instance that contains the executed Effect and a flag denoting whether this was an undo operation or not. In order to further distinguish the TriggerStates a Combination actually receives there is a Java generics type parameter denoting the class of Effect contained, allowing a Combination to specifically state what notifications it needs to receive instead of getting executed for every potentially irrelevant Effect. Additionally, this TriggerState utilizes the ability to specify a dynamic key class as introduced in Section 4.1.4. The default implementation would provide the TriggerState class as the key, here the class of the contained Effect is taken as a replacement. Without that feature it would not be possible to differentiate between the various `EffectTriggerState` instances that exist for the diverse types of Effects - in that case there could only be one TriggerState at a time. The major benefit of storing previous instances would disappear, namely letting Combinations access the current situation without having to cache this information itself.

### 5.1.4 ModelChangeTrigger

Every resource handled by a GMF editor actually consists of two files. The model file contains the semantic information based on the domain-specific EMF meta model, while the diagram file holds the notation of the visual representation. Any change to one of these two is of particular interest to the view management. By monitoring model or diagram modifications various improvements of the user experience can be implemented, for instance the automatic re-calculation of the layout after an invalidating operation such as adding a new edge to a graph. The `ModelChangeTrigger` receives these notifications through the Eclipse `ResourceSetListener` mechanism. It is part of the KIELER core model package that is included in virtually all KIELER installations.

In order to let Combinations easily distinguish the different types of resource changes this Trigger can generate either a `ModelChangeState`, a `DiagramChangeState`, or both. This distinction only serves as a simplification for Combination developers, there is no functional difference between the two TriggerStates. Typically a Combination would only require exactly one kind of change notification, this artificial splitting up facilitates the correct identification by Combinations. Both TriggerStates provide the `DiagramEditor` where the modification occurred and the `ResourceSetChangeEvent` describing the changes. By passing on the entire event a Combination can utilize the default Eclipse `NotificationFilter` mechanism to easily sift through the data.

### 5.1.5 StateActivityTrigger

The `StateActivityTrigger` serves as a link between the KIELER Execution Manager and the view management during the simulation of a SyncCharts diagram. KIEM takes care of the actual stepping through the model and provides a list of active states along with the transitions taken in the most recent step. In order to access this information the SyncCharts view management package contains a data component that is registered with the execution manager. The data component translates the list into actual `EObject`s and additionally looks up a user-definable number of previous so-called history steps in the KIEM data pool. This set of states and transitions is stored in `ActiveStates` instances along with the `DiagramEditor` that contains the simulated model. By registering itself for this TriggerState a Combination can visualize the execution of a SyncCharts diagram.

## 5.2 Effects

### 5.2.1 UndoEffect

The `UndoEffect` is a vital but hidden part of the view management. In order to simplify the `EffectsWorker` for the handling of both executions and undos of Effects this serves as a wrapper. It takes another Effect as a parameter and calls its `undo` method when executed. For a Combination developer there is no point of contact with this Effect. When merged the `UndoEffect` checks whether the other Effect is the same object as the one supposed to get reverted. If that check yields true then the other Effect is discarded because there would be no point in executing it while it already is queued to be undone. That would only create additional processing overhead and potentially produce undesired flickering graphics.

### 5.2.2 LayoutEffect

One of the first KIELER projects was the Infrastructure for Meta Layout. The `Layout-Effect` integrates its features into the view management environment and is available with the KIML package. Instead of knowing how to invoke the layout algorithms directly a Combination developer now only needs to schedule this Effect for the diagram that shall be evaluated. There is a wide variety of parameters on hand to modify the Effect's behavior. Most importantly the Combination can choose whether the diagram should be zoomed to fit the editor's viewport together with the application of the new layout. In order to maintain the user's mental map throughout the rearrangement any zooming out is performed before the layout animation is run. This way no part of the motion is hidden from view. Conversely, zooming in closer takes place subsequently to avoid covering any diagram elements that would later move into the visible area. As a result, the entire layout animation is visible in every situation if the zoom to fit option is enabled.

Calculating and applying a new layout takes a considerable amount of time. Hence it is of utmost importance to avoid redundant executions of the `LayoutEffect`. To achieve this the Effects merging mechanism is utilized. If there are two `LayoutEffects` working with the same diagram on the `EffectsWorker` queue the earlier one will be discarded. However, a `LayoutEffect` can have a target `EObject`, in which case only the corresponding element along with its children and line of ancestors are processed. Quietly discarding such an Effect would modify the intended result. In order to prevent this the `merge` method takes both targets and computes their common ancestor. The resulting Effect supersedes both individual layouts and can therefore be used safely as a replacement.

### 5.2.3 HighlightEffect

This Effect changes the way the visual representation of a model object looks. In order to restore the previous values when the `HighlightEffect` is undone every modification is accompanied by the cached old value. Most importantly it changes the foreground color, optionally retaining the font color of contained labels. Often the thickness of a shape's outline is too thin to clearly make out the new color on all kinds of devices, especially with projector-based demonstrations. To overcome this problem the line width can be increased while this Effect is activated. Using the color as the primary indication for highlighted objects can be troublesome when using exported images in publications because these might be restricted to grey scale printing. Hence the `HighlightEffect` supports changing the outline style to either dotted or dashed to improve the conspicuity compared to regular shapes for those cases. Additionally, the background color can be modified to add another distinguishing attribute, significantly increasing the number of permutations that can be told apart from each other easily. This Effect is available to every project utilizing KIELER technology because it resides in the core feature.

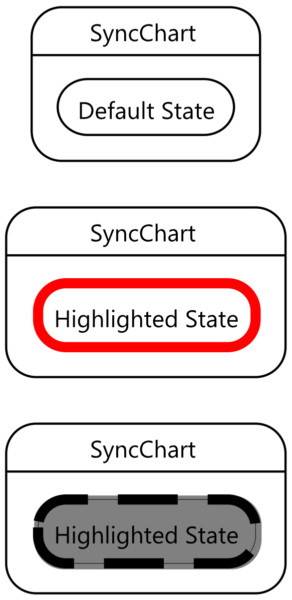


Figure 5.1: Various styles of highlighting a SyncCharts state.

Figure 5.1 showcases some results of applying the `HighlightEffect` to a SyncCharts state. On top the inner state is shown without any modification, the middle one has received a thicker, red outline, and the bottom state displays the black & white mode with a dashed outline and grey background.

### 5.2.4 CompartmentCollapseExpandEffect

Many types of diagrams are arranged in a hierarchical tree structure where one node may contain a number of other nodes. Graphically this is represented by a large compartment within the outer macro node. One major problem that arises when working with larger models is the inability of traditional zooming and panning to retain both the overview of the entire system and detailed insight into the inner levels of hierarchy. Zooming in hides large parts of the diagram, zooming out makes labels unreadable. By collapsing compartments that currently are not of interest to the user a lot of valuable screen real estate can be saved and put to better use. For instance, during a SyncCharts simulation any inner states of an inactive state are less relevant. Hence hiding them from view by collapsing their container compartment is a good way of focusing on the active states and their neighbors. The considerable difference in space taken up by collapsed states compared to their expanded counterparts is shown in Figure 5.2.

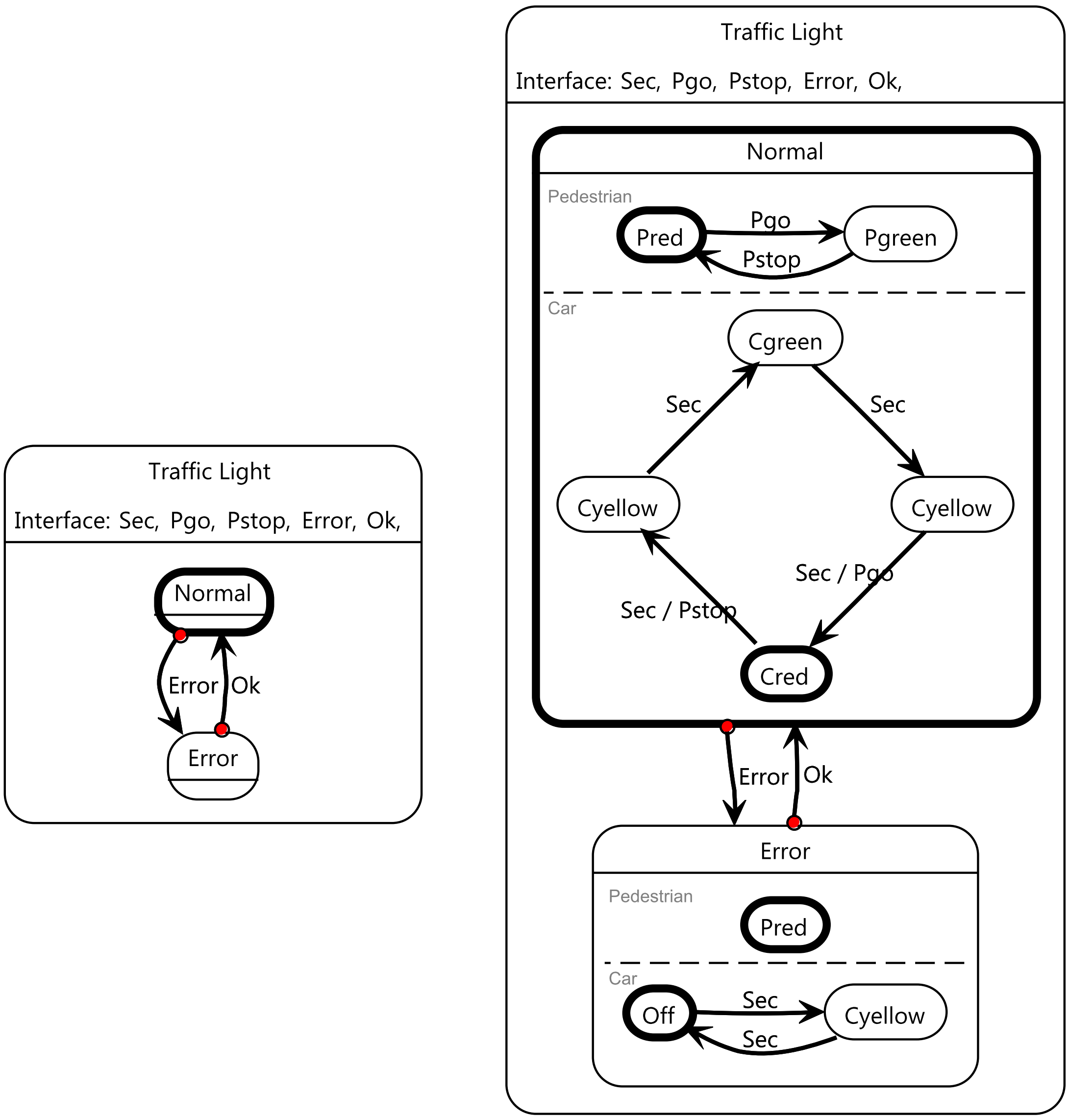


Figure 5.2: Collapsed and expanded SyncCharts states.

The `CompartmentCollapseExpandEffect` is a prime example for the `EffectTrigger`. Typically the previous layout becomes invalid and needs to be recalculated after a compartment has been collapsed or expanded. This task is performed by a Combination registered for instances of `EffectTrigger<CompartmentCollapseExpandEffect>`. Multiple calls to the `LayoutEffect` are avoided by the merging mechanism for Effects. `CompartmentCollapseExpandEffects` discard any previous Effect on the queue when merged as long as the compartment to collapse is identical. Removing undos for the same target from the queue suppresses changing the state of a compartment and going back to the previous condition in quick succession.

### 5.2.5 ArrowEffect

The `ArrowEffect` is specific to SyncCharts and draws a colored arrow from one graphical element to another. When merged it consumes other `ArrowEffects` targeting the same pair of `EObjects` along with any undos for that pair. Without that approach arrows could flicker off and on when a Combination repeats the previous state of its Effects in a subsequent execution.

## 5.3 Combinations

### 5.3.1 LayoutCombination

The `LayoutCombination` in the KIML package receives both the `SelectionState` and the `ButtonState`. Nevertheless it only goes into action when a KiVi button has been pressed, the most recent selection merely serves as a potential parameter to the layout algorithm. This fact is checked by comparing the sequence numbers. If the `ButtonTrigger`'s sequence number is greater then this TriggerState is responsible for the firing of the current execution. After establishing that order the Combination checks whether the pressed button belongs to the layout or not. If it does, the preference values for whether zoom to fit should be performed and whether a progress bar should be displayed during the layout computation are retrieved and passed on to a new `LayoutEffect` instance. Finally this Effect is scheduled using the shortcut methods provided by the abstract Combination implementation.

### 5.3.2 LayoutAfterCollapseCombination

This Combination works in tandem with the `CompartmentCollapseExpandEffect`. It listens to the appropriate `EffectTrigger` and schedules a new `LayoutEffect` every time a compartment has been collapsed or expanded. Redundant layout calls are removed automatically by the Effects merging mechanism.

### 5.3.3 ShowHierarchyCombination

Many types of models allow for deeply nested hierarchy. This may become confusing, especially when parts of the diagram are hidden by the current zoom level. The `ShowHierarchyCombination` uses a large number of `HighlightEffects` to paint every level of hierarchy in a unique color by stepping through the entire hue wheel. As a result the root level starts out with red, its offspring successively are painted in different shades of yellow, green, blue, and pink. Finally the deepest leaf completes the circle with another tone slowly approaching red again. This colorful mode can be turned on by a bespoke button in the KIELER toolbar, again using the `ButtonTrigger` to simplify the registration process. The colorful result displayed in Figure 5.3 may be used as an inspiration for future Combinations to highlight the different levels of hierarchy in a better way.

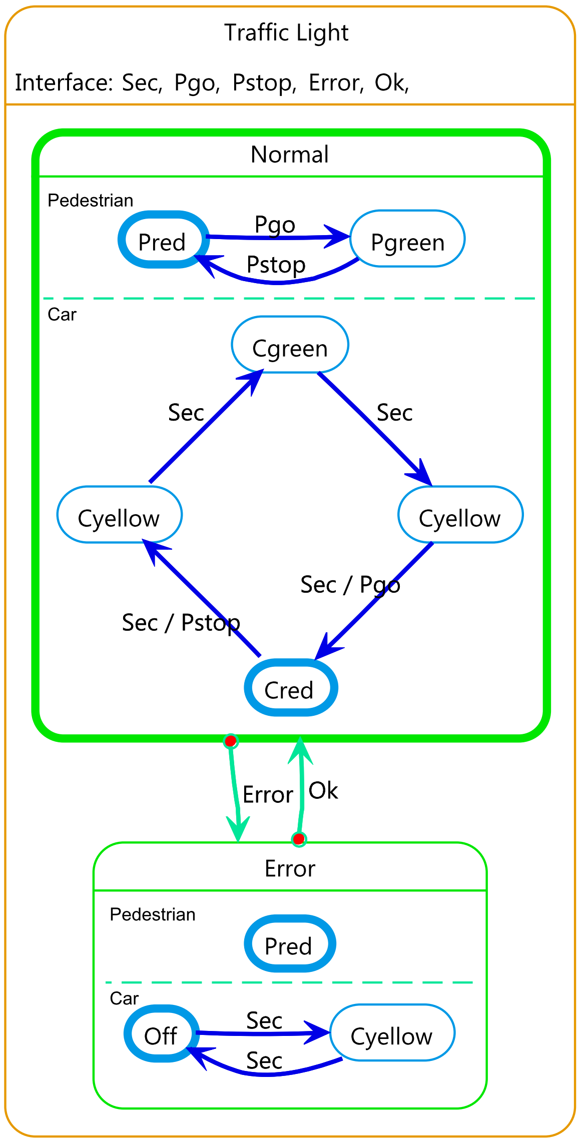


Figure 5.3: A SyncCharts diagram using the hierarchy Combination.

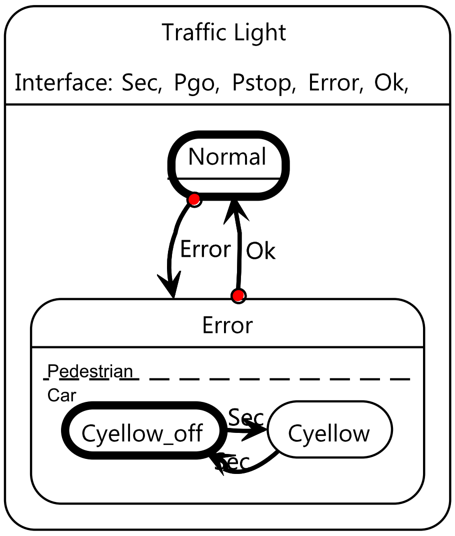


Figure 5.4: Invalid layout after changing a state name.

### 5.3.4 LayoutAfterModelChangedCombination

Almost every modification the user does to a diagram will invalidate its layout, whether it is adding a new node or simply extending a label that inadvertently will require more surrounding free space. The automatic layout algorithms provided by the KIELER project already relieve the user of having to move objects around with the mouse, however that still requires manually calling those algorithms. The `LayoutAfterModelChangedCombination` receives `ModelChangedStates` as well as `DiagramChangedStates` and analyzes their contained event. If that event is determined to potentially invalidate the existing layout the Combination schedules a new `LayoutEffect` for the modified SyncChart. Due to this the user automatically can experience the benefits of frequently recalculated layouts.

Figure 5.4 shows an example with the `LayoutAfterModelChangedCombination` deactivated. The user has changed the name of the left state from “Off” to “Cyellow_off”. As a result the size of that state has increased and it has severely deformed the two transitions connected with it. Using the Combination in Figure 5.5 avoids this problem by monitoring the change and automatically scheduling a `LayoutEffect`.

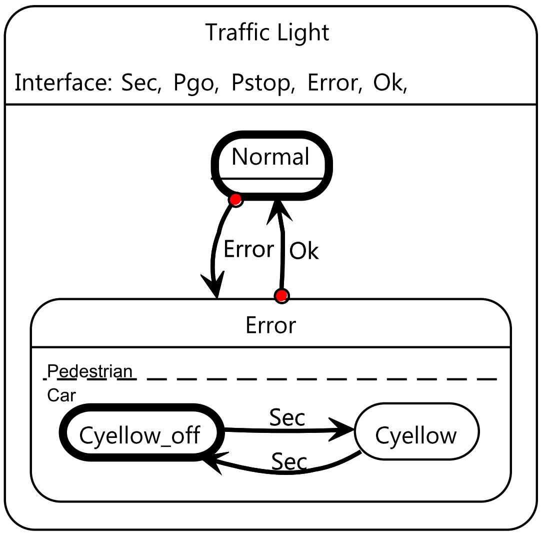


Figure 5.5: Recalculated layout triggered by the view management.

### 5.3.5 SignalFlowCombination

A commonly criticized aspect of SyncCharts is the absence of any visible notion of signal dependencies. Some transition may be taken because a broadcast signal has been emitted in an entirely different part of the diagram. For large models it is virtually impossible to keep track of these complex relationships. The `SignalFlowCombination` addresses this problem by using the `ArrowEffect` to visualize the flow and dependencies of signals. If there is a signal emitted by transition A and tested by transition B then there will be an arrow originating at transition A and pointing at transition B, effectively stating that taking transition A may cause transition B to be taken. Figure 5.6 gives a simple example demonstrating this feature. In large diagrams the amount of arrows can quickly become overwhelming. Their number can be reduced and focused by selecting certain model elements, the Combination will then only display the signal dependencies that concern any chosen object. The exact behavior depends on the types selected. For selected signals simply all arrows will be shown that address this signal. Selecting a transition will display all incoming and outgoing arrows, denoting possible reasons why this transition may be taken and potential follow-up transitions as a result of a signal emission. If the list of selected elements contains more than one member then the arrows for each object are drawn together similar to the union of a number of sets. This filtering mechanism is shown in Figures 5.7 and 5.8, here the total number of signal dependencies is far too great to be drawn all at once. The second image restricts this to only the signals related to a single selected transition.

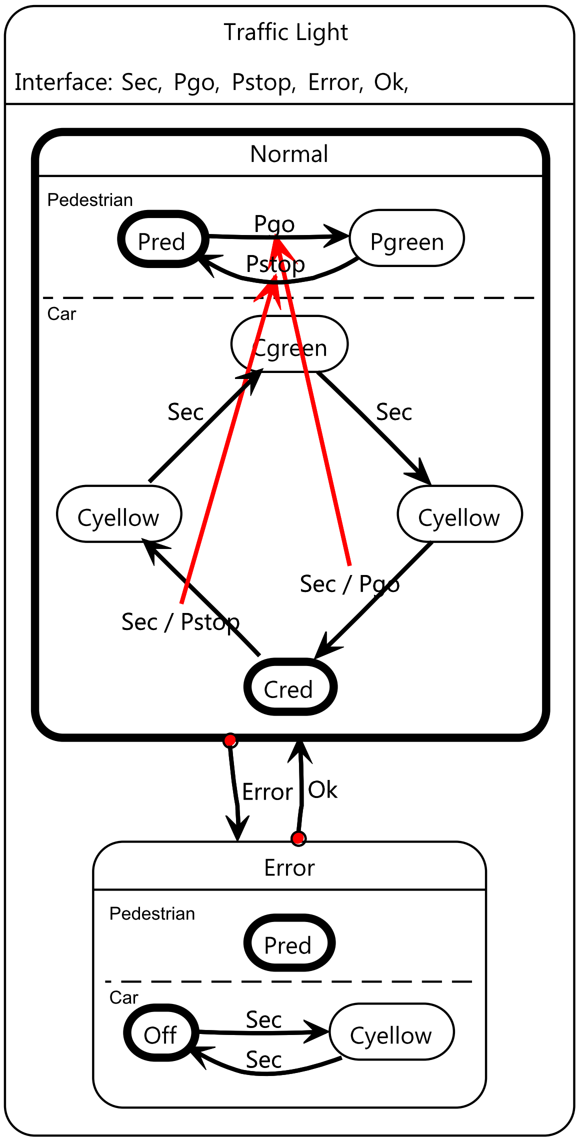


Figure 5.6: Displaying the flow of signals.

### 5.3.6 SyncChartsCombination

SyncCharts can be simulated using the KIELER Execution Manager. Visualizing this is an ideal application for the view management. The `ActiveStates` TriggerState is providing a list of currently active states as well as a couple of lists for the most recent history steps. The `SyncChartsCombination` uses this in two ways.

Each of these states is colored in a distinctive color with the `HighlightEffect`. Active

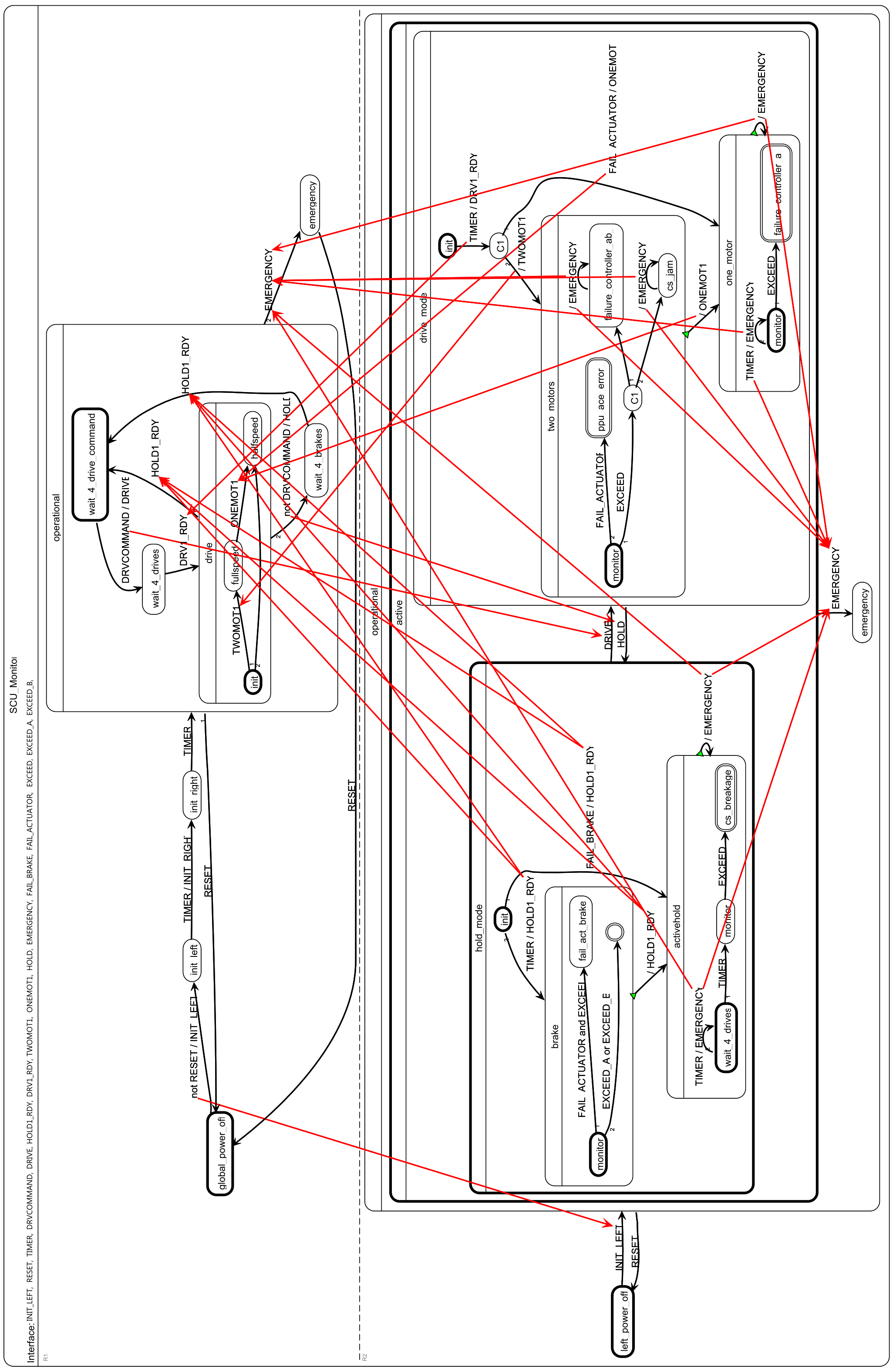


Figure 5.7: An overwhelming number of signal flow arrows.

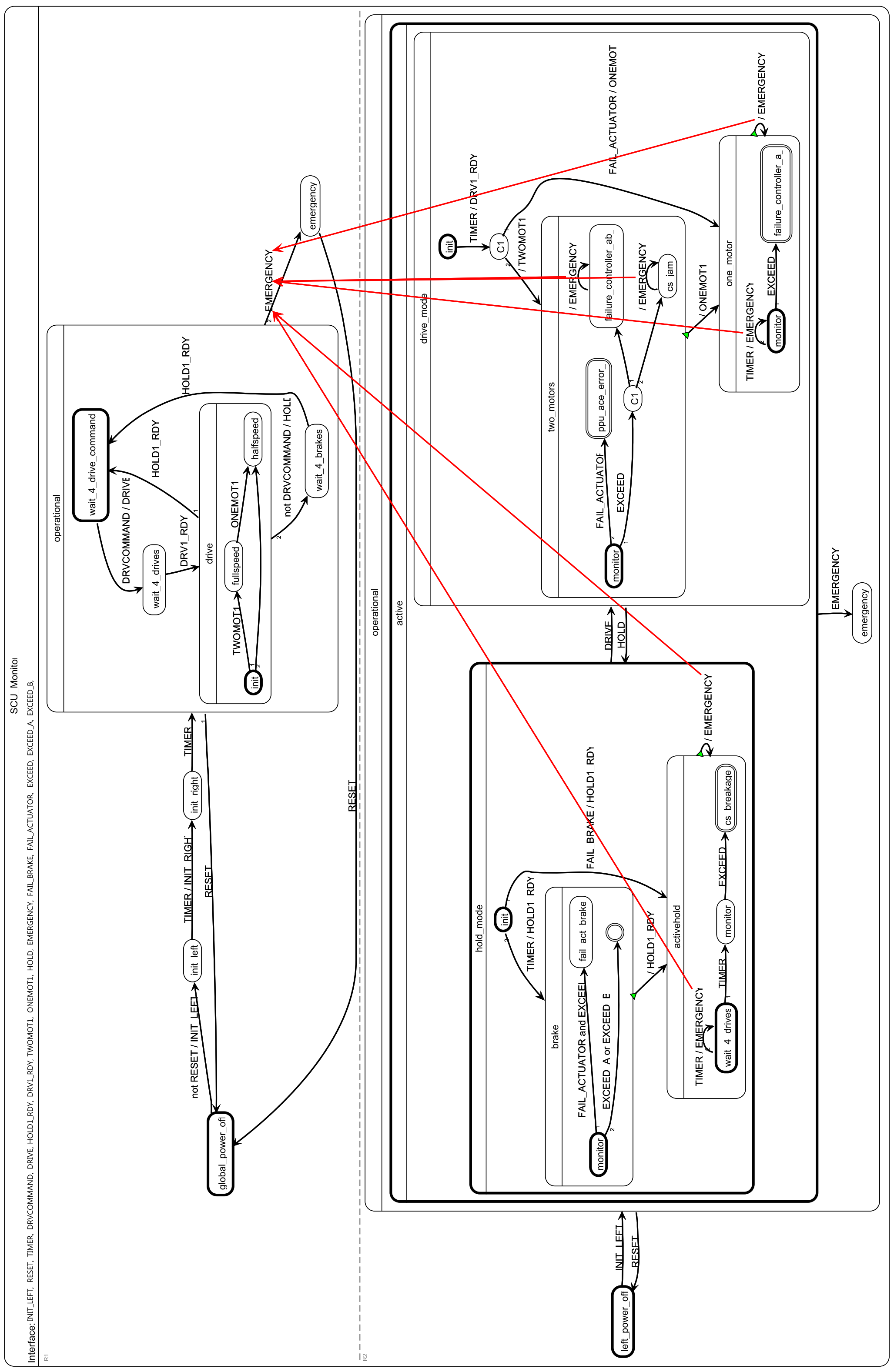


Figure 5.8: Filtering the signal flow arrows for a single transition.

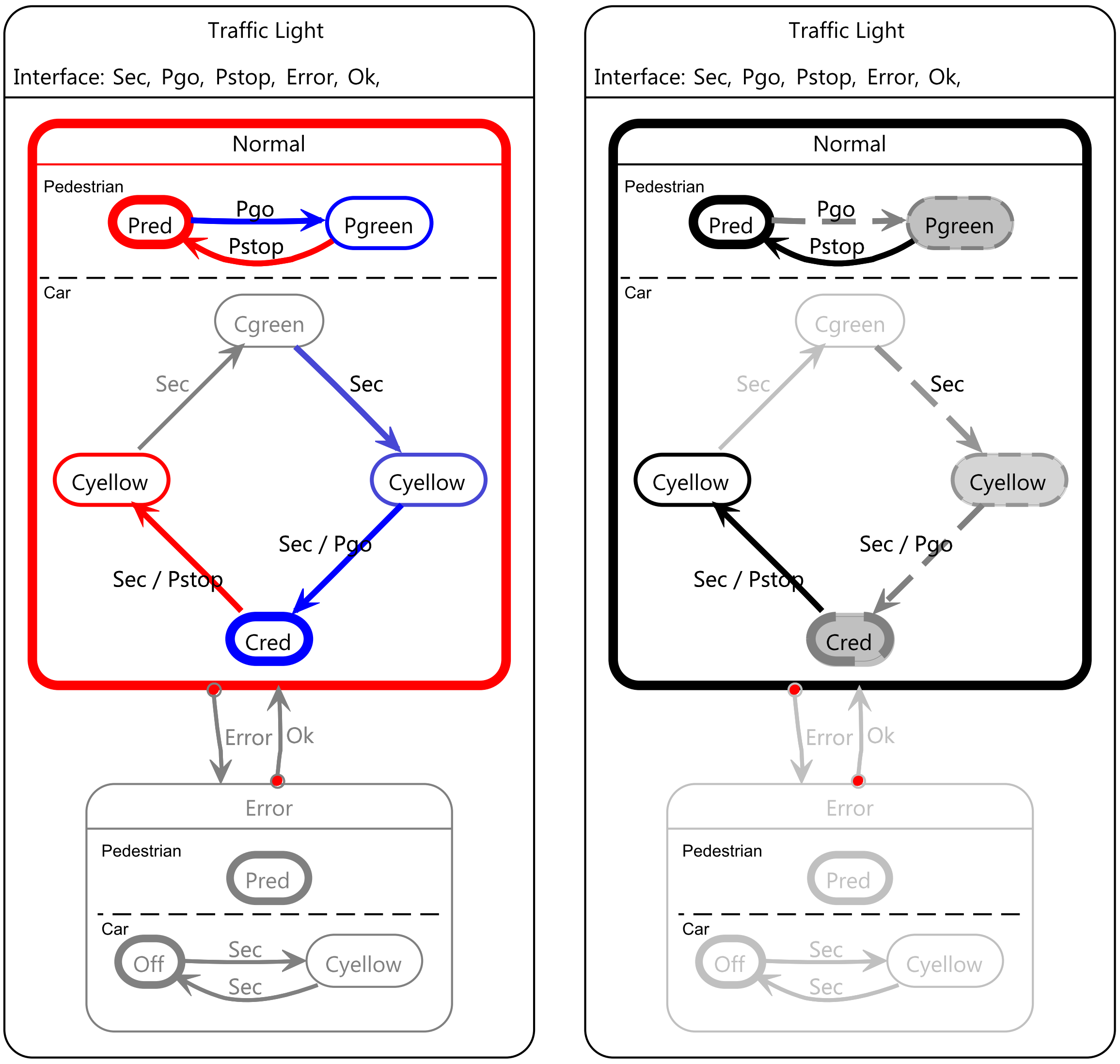


Figure 5.9: Simulation visualization both in color and black & white mode.

states are painted red by default, the most recent history states that have just become inactive are painted blue by default. The remaining set of history states is gradually faded towards the color of inactive states, by default this is grey. The older a history state is, the more grey and less blue it becomes. On top of that the user can choose to give these three types of state - active, history, and inactive - a background color that is processed in the same color-fading way as the foreground color. In order to show the colors more clearly the `HighlightEffect` is set to increase the width for both active and history states. However there are cases where the rich use of colors is inappropriate, such as when printing in grey scale. For this scenario the `SyncChartsCombination` comes with a black & white mode. Enabling this will change the line style for history states from solid to dashed, making the distinction between the different conditions of states much easier. Together with a selection of black and grey background and foreground colors the user can tailor the look of the diagram to fit the specific printer. For instance, active states could be drawn with black outlines and white backgrounds. History states might retain the same colors because the dashing already serves as a distinction. Inactive states could be drawn with black outlines and grey backgrounds, together with the lower line width these will be sufficiently different from active states. If the printer can produce distinctive shades of grey then the inactive state outlines could be chosen as dark grey to further facilitate the distinction. In that case the fade from white history backgrounds to grey inactive backgrounds would also become visible. A comparison of the color and black & white mode is shown in Figure 5.9.

The second method employed utilizes the `CompartmentCollapseExpandEffect` and has been coined Focus & Context. Maintaining an overview of a large diagram as well as keeping track of intricate details is next to impossible. Yet, during simulation inactive parts of the diagram typically are of less interest to the user. This fact is exploited here by collapsing inactive macro states, hiding their contained hierarchy levels. This reduces the visual clutter and allows for a greater zoom level, focusing the user's attention on the active and history states. The context of those states is still shown because only the inner parts of inactive states are hidden. The simple traffic light example already shown before is displayed using Focus & Context in Figure 5.10. The inactive Error state is collapsed, hiding its children and thus taking up a lot less space.

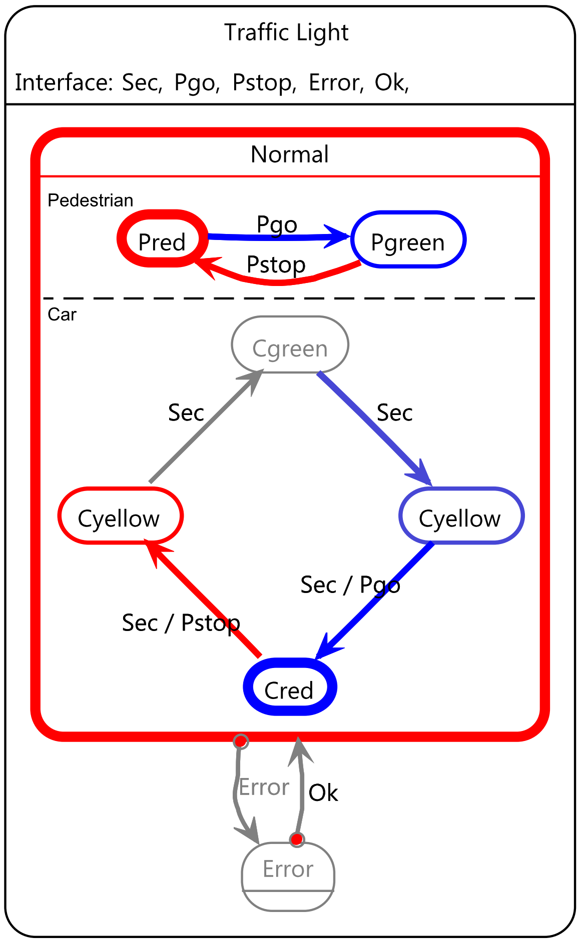


Figure 5.10: Focus & Context reduces the space taken up during simulation.

### 5.3.7 HighlightSelectedTransitionsCombination

This is a small SyncCharts helper Combination to simplify working with transitions and their labels. In many scenarios there are multiple labels and transitions close together, making it hard to mentally associate one with the other. Any selected transition and its label are highlighted in blue by the `HighlightSelectedTransitionsCombination`. Now the user can click on one and immediately spot the other by the new color. In Figure 5.11 the transition labels have been positioned in an unfortunate cluster in the middle. Selecting a transition will paint it and its label blue, making their relationship easily recognizable.

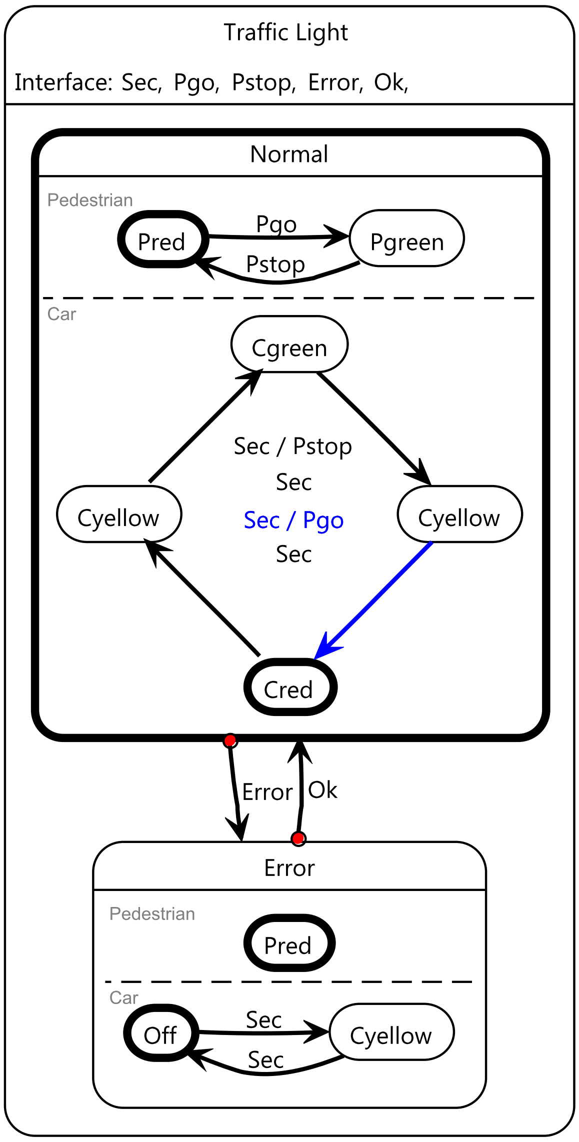


Figure 5.11: Highlighting the selected transition and its label.

```java
public class HighlightSelectedTransitionsCombination
    extends AbstractCombination {

  public void execute(final SelectionState selection) {
    if (selection.getDiagramEditor() instanceof
        SyncchartsDiagramEditor) {
      for (EObject selected : selection.getSelectedEObjects()) {
        if (selected instanceof Transition) {
          HighlightEffect e = new HighlightEffect(selected,
              selection.getDiagramEditor(),
              ColorConstants.blue, true);
          e.setChangeWidth(false);
          schedule(e);
        }
      }
    }
  }
}
```

Table 5.1: The entire Combination source code highlighting selected transitions.

The `HighlightSelectedTransitionsCombination` is a great example of how simple Combinations benefit from the abstract implementation in the background. Table 5.1 shows its entire source code. The `execute` Method registers itself for instances of the `SelectionState`. When it is called by the view management framework it only needs to check whether the editor where the change of selection happened actually works with SyncCharts. In this the case then every transition in the list of selected objects is highlighted with the existing `HighlightEffect`.

# 6 Conclusion

## 6.1 Summary

The KiVi project set out to design and implement a framework supporting the dynamic modification of the visual appearances for graphical diagrams. It defines four core component types - Triggers, TriggerStates, Effects, and Combinations - that can be extended by any foreign Eclipse plug-in. Additionally there is a robust concept concerning the issue of concurrency in a complex application. Effects and Combinations each get their own dedicated thread, ensuring predictable behavior without forcing the developer to utilize error-prone techniques such as the Java `synchronized` keyword.

A key indicator of success for any piece of software is its robustness and reliability. KiVi fully encapsulates any foreign code, effectively keeping the system running even if uncaught exceptions occur due to programming errors in external classes. However, there still is room for improvement. For instance running certain view management features such as the simulation visualization with highlighting, collapsing, and automatic layout on large diagrams will cause significant performance issues on Mac OSX. Apparently the native Java implementation drawing low-level Eclipse graphics is considerably slower compared to similarly powerful machines running either Linux or Windows.

Apart from providing the central view management framework a major task for this master thesis was to create several applications using KiVi to test and demonstrate its capabilities. A whole range of Triggers and Effects allows future programmers to write higher-level Combinations without digging deeply into the inner workings of Eclipse. Various Combinations that are part of this project already are in use by other KIELER developers and users. Most prominently everyone is subconsciously benefiting from the implicit automatic calls to the KIML layout algorithms whenever there is a modification to a SyncCharts diagram that invalidates the current layout. New features such as the set of history states in SyncCharts simulations help users follow the control flow through the diagram even when many transitions are taken in a single step. Due to the black & white mode exported diagrams can finally be included in papers without the tedious task of explaining what state should be imagined to have been printed in which color by the reader.

Finally, numerous projects have already begun to use the view management framework for their own purposes. For instance the in-development UML State Machine Simulation and Model Checking with Maude project[1] comes with its own visualization for both simulations and model checking traces. This is a new Combination that has been tailored to the intricacies of the Papyrus UML editor[2]. It re-uses the existing `HighlightEffect` to colorize states and modify their outlines.

## 6.2 Future Work

There are various entry points for future projects extending and improving the view management framework. Some included Combinations are tailored to work with SyncCharts, yet there are many more editors to consider such as the KIELER Aspect-Oriented Modeling project (KAOM)[3]. Creating a set of Combinations answering the specific needs of that editor would significantly improve its user experience.

Currently writing a new Combination always entails coming into contact with real Java code, even though the abstract implementations described in Section 4.1.6 already simplify this chore as much as possible. Ideally this process could be further lifted from low-level programming onto some type of Combinations scripting language. Essentially such a script would comprise of the inner parts of the current `execute` method, testing for conditions and scheduling Effects accordingly. A very basic language like that might even be interpretable at run time, allowing users with little programming experience to write or modify Combinations from within their ongoing Eclipse instance.

[1]`http://rtsys.informatik.uni-kiel.de/trac/kieler/wiki/Projects/UMLSim`
[2]`http://www.papyrusuml.org`
[3]`http://rtsys.informatik.uni-kiel.de/trac/kieler/wiki/Projects/KAOM`

# Bibliography

[1] Torsten Amende. Synthese von SC-code aus SyncCharts — Ein compiler für SyncCharts. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, May 2010. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/tam-dt.pdf`.

[2] Charles André. SyncCharts: A visual representation of reactive behaviors. Technical Report RR 95–52, rev. RR 96–56, I3S, Sophia-Antipolis, France, Rev. April 1996.

[3] Nils Beckel. View Management for Visual Modeling. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, October 2009. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/nbe-dt.pdf`.

[4] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, Jan 2003.

[5] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software—Practice and Experience*, 30(11):1203–1234, 2000.

[6] Object Management Group. Unified Modeling Language, Superstructure version 2.3. `http://www.omg.org/cgi-bin/doc?formal/2010-05-05.pdf`, May 2010.

[7] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[8] Steffen Jacobs. Konzepte zur besseren Visualisierung grafischer Datenflussmodelle. Student resarch project, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, February 2007. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/sja-st.pdf`.

[9] Christian Motika. Semantics and execution of domain specific models — KlePto and an execution framework. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2009. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/cmot-dt.pdf`.

[10] Steve Northover and Mike Wilson. *SWT: The Standard Widget Toolkit*, volume 1. Addison-Wesley, July 2004.

[11] Inc. Object Technology International. Eclipse platform technical overview. `http://www.eclipse.org/whitepapers/eclipse-overview.pdf`, February 2003.

[12] Steffen Prochnow and Reinhard von Hanxleden. Statechart development beyond WYSIWYG. In *Proceedings of the ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS'07)*, volume 4735 of *LNCS*, Nashville, TN, USA, October 2007.

[13] Arne Schipper. Layout and Visual Comparison of Statecharts. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2008. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/ars-dt.pdf`.

[14] Matthias Schmeling. ThinKCharts — The thin KIELER SyncCharts editor. Student research project, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, September 2009. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/schm-st.pdf`.

[15] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF Eclipse Modeling Framework.* Eclipse Series. Addison-Wesley, Pearson Education, 2$^{nd}$ edition, 2009.