View Management (KIVi)

This project is currently not developed further.

Project Overview

Related Publications:

- Hauke Fuhrmann and Reinhard von Hanxleden. On the Pragmatics of Model-Based Design. In *Foundations of Computer Software. Future Trends and Techniques for Development—15th Monterey Workshop*, Revised Selected Papers, vol. 6028 of LNCS, p. 116–140, Springer, 2010. (pdf) The original publication is available at link.springer.com.
- Hauke Fuhrmann and Reinhard von Hanxleden. Taming Graphical Modeling. In *Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems* (MoDELS'10), vol. 6394 of LNCS, p. 196–210, Springer, 2010. (pdf) The original publication is available at link.springer.com.

Related Theses:

- Nils Beckel, *View Management for Visual Modeling*, October 2009. (pdf)
- Martin Müller, *View Management for Graphical Models*, December 2010. (pdf)
- Hauke Fuhrmann, *On the Pragmatics of Graphical Modeling*, 2011. Disputation: May 5th, 2011 (pdf)

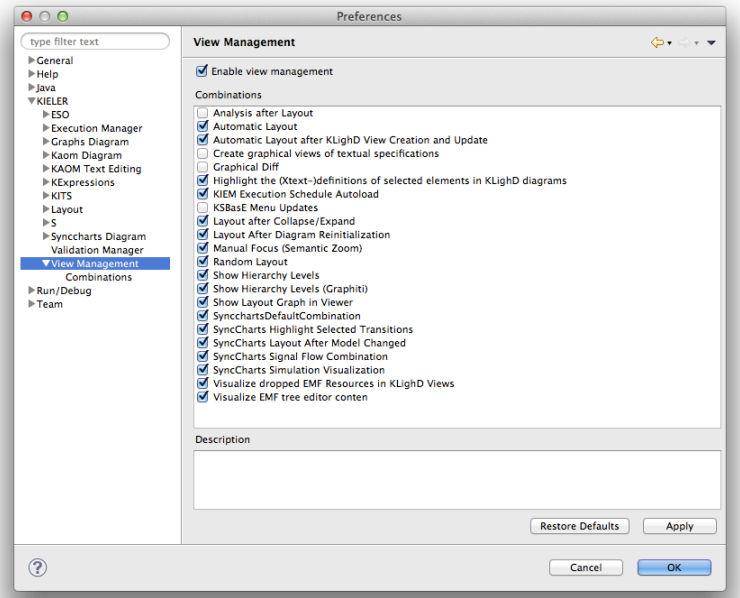
Introduction

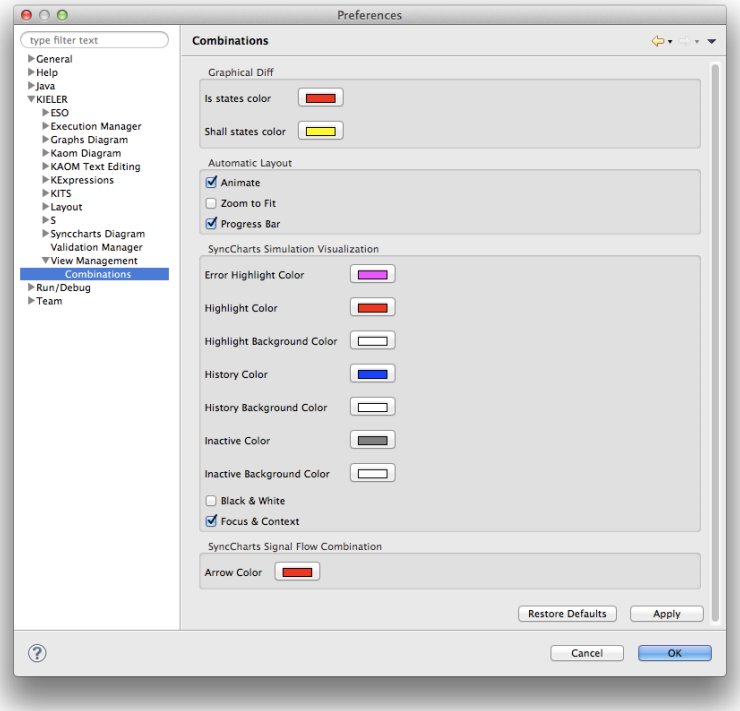
KIELER View Management (KIVi) is a high-level description engine for dynamic visualizations in diagrams (e.g. for graphical modeling). Its main driver is the availability of automatic layout as provided by KIML. It implements user interaction that builds upon automatic layout and therefore unfolds the full potentials of automatic layout. Different use-case examples are:

- Navigating in complex models using "Focus&Context"
  - Show currently "interesting" elements in the Focus with full detail
  - Show other elements as the Context in reduced detail (e.g. compartments collapsed, labels hidden)
- Use execution/simulation results for interactive debugging visualization of diagrams
- Perform editing by executing model-to-model transformations on the domain model while view management updates the diagram view

User Guide

KIVi provides two preference pages to configure the view management. The first page lets you activate the view management itself and any combination you need. The second page shows all properties of all combinations registered with KIVi.





Enabling View Management Combinations will start the corresponding interaction paradigm automatically. It might react on internal Eclipse or Simulation events or it might wait for buttons pressed by the user. The result usually alters the currently visible diagrams.

Developer Guide

The core view management uses three interacting components: Triggers, Combinations, and Effects. Implementing a new use-case for view management usually stays on a rather high-level abstraction of a *Combination* which specifies, *under what conditions* (triggers) a diagram should be displayed *in what way* (effects), e.g. in what level of detail. *Triggers* and *Effects* are low-level implementations of specific events resp. graphical effects in a diagram that Combinations work with. As KIELER provides a set of pre-defined triggers and effects (see list below), a developer might only work at the level of an Combination. Only if the provided triggers and effects are not enough for your combination, you need to delve a little bit deeper into Eclipse to extend KIVi by your own triggers/effects.

Examples

See the running example ManualFocusCombination.java source code as an example how simple implementation of a new view management use-case is.

Find more minimal examples in the

- Unit Tests with many minimal examples

Triggers

A Trigger is a low-level observer that notifies the view management about certain events. Each Trigger has an associated TriggerState class that handles these events and presents them as the current state to the combinations.

Every Trigger needs to implement the ITrigger interface, most conveniently by extending AbstractTrigger. Analogously the TriggerStates need to implement ITriggerState or extend AbstractTriggerState. Extending the abstract classes is the recommended method to avoid re-implementation of some core features: Using the abstract implementation, a new Trigger only needs to call its own trigger(ITriggerState) method with a new instance of its TriggerState representing the new event whenever a new event needs to be passed to the view management.

Synchronization

It may happen that Triggers fire way too often and flood the system with too many events. In such case Combinations are also called very often and will flood the EffectsWorker queue with too many effects and the effects queue can overflow, resp. effects will not be executed in a reasonable time after the causing trigger.

If it is known in advance that a trigger might occur often, it can be synchronized with the effects queue. Call AbstractTrigger.synchronizedTrigger (ITriggerState) instead of the normal trigger method in order to block the current thread until all effects have been executed that have been caused by this triggering. This way backpressure can be induced from effects to the triggers.

Technically the synchronized step calls wait() on the corresponding TriggerState. The CombinationsWorker thread puts all effects resulting from a Combination onto the EffectsWorker's queue. Afterwards it also puts an UnlockEffect onto the queue with the corresponding TriggerState as parameter. The EffectsWorker thread eventually will take the UnlockEffect from the queue after all effects have been executed and will notifyAll() who listen to the given TriggerStates. This guarantees time-synchronization between the three involved threads, i.e. the Trigger will be blocked until the EffectsWorker has released the old TriggerState. Note that this may deadlock, e.g. if the Trigger executes in the UI thread, i.e. it blocks the UI thread and one effect also executes in the UI thread (e.g. LayoutEffect). Then this effect will wait for the Trigger and the Trigger waits for completion of all effects -> Deadlock. As many effects could arbitrarily work on the UI thread, avoid using synchronizedStep() in a Trigger that calls from the UI thread.

Combinations

Combinations bind Triggers and Effects together: A Combination is notified of updated TriggerStates that concern the Combination, and the Combination launches appropriate Effects to visualize these for the user.

Similarly to Triggers there is both an interface ICombination and an abstract class AbstractCombination, either could be used to create a new combination. However, it is strongly recommended to use the abstract superclass because a lot of convenience is already built into it. That way you only need to implement an execute(TriggerState1, TriggerState2, ...) method that takes all TriggerState types you need as a parameter, all details such as making sure the corresponding Triggers are activated is already taken care of. This way the combination developer only needs to implement a very small and high-level part in KIVi as marked in the above sequence diagram.

The AbstractCombination provides a very convenient Effect handling mechanism. In order to launch an Effect you only need to call schedule(theEffect), the abstract implementation will take care of the rest. Specifically, all Effects are automatically undone when the entire combination is undone (e.g. disabled). Effects can be easily recorded to be explicitly undone, e.g. when one execution wants to undo the effect of earlier executions. Just call undoRecordedEffects() in an execute method. This removes the need for any Effects housekeeping within the Combination, for example remembering which Effects have been executed in the past and which need to be undone now. You simply schedule the Effects necessary to produce the desired state, independently of any past Effects - the Effects merging mechanism will take care of any redundant operations.

Combinations can contribute parameters to the KIVi Combinations preference page by implementing the static method CombinationParameter[] getParameters(). As of now, the preference page supports Strings, Integers, Floats, Doubles, Booleans and RGB color values. These parameters will automagically appear on the Combinations preference page as posted in the User Guide above.

In order to let KIVi know about your combinations you need to create an extension for the Combinations extension point.

Effects

An Effect applies some visual change to the diagram, for example it might perform automatic layout. All Effects are created and scheduled by Combinations, their execution is done from within the Effects worker thread.

New Effects are implemented by implementing IEffect or preferably by extending AbstractEffect. Then you only need to implement the execute() method, and override undo() if your Effect can be undone.

Many Effects are mergeable, for example if two Layout Effects are scheduled then one layout run would suffice. In order to support merging your Effect needs to let isMergeable() return true and implement merge(IEffect). The merge() method returns null by convention if the two Effects can not be merged, and it returns the newly merged Effect (usually itself with some modifications) if the two Effects can be merged - then the other Effect is automatically discarded. Correct merging of Effects is particularly important for the automatic scheduling/undoing mechanism in the AbstractCombination. It will often undo an effect right after scheduling its execution, and give programmer convenience in the Combination in return.

Implemented Features

Follow the Javadoc links to see the corresponding *package* for the triggers, effects and combinations. It also gives a hint, which *plug-ins* to load in order to get access to the classes in your own plug-in.

Trigger(States)

- SelectionTrigger - Contains the most recent list of selected EObjects and the containing DiagramEditor
- ButtonTrigger- Generic TriggerState for KIVi buttons registered using the ButtonHandler. Buttons can be defined
  - using the standard Command extension point of eclipse and by using the ButtonHandler as the handler, or
  - defining a button programmatically, e.g. in the constructor of a Combination, using the KiviMenuContributionService. See an example Combination in KiviMenuContributionDemoCombination.java
- EffectTrigger - Triggered when a KIVi Effect has been executed
- ModelChangeTrigger - Contains the most recent change to a semantic model
- ModelChangeTrigger.DiagramChangeState - Contains the most recent change to a GMF diagram model
- ModelChangeTrigger.ActiveEditorState - Information about the active editor (not necessarily containing a diagram)
- DiagramTrigger - Information about the active diagram (e.g. in the active editor or view), used instead of ActiveEditorState with convenient access to diagram and semantic model

Syncharts and Papyrus UML StateMachine specific

- StateActivityTrigger - receives the most recent n steps of active states from the KIVi-KIEM data component during simulation of Syncharts or Ptolemy UML StateMachines

Effects

- UndoEffect - Pseudo-Effect used to simplify undoing Effects. The only point of contact as a developer is when merging Effects, the other Effect passed may be an UndoEffect containing the actual Effect
- MenuItemEnableStateEffect - Change the enabled state of a button defined with KVi
- LayoutEffect - Performs automatic layout with various options
- SetOptionsEffect - Sets layout option values for a specific model element
- HighlightEffect - Provides different methods of highlighting an EObject: Change its color, line width, or line style without changing the notation model
- CompartmentCollapseExpandEffect - Collapse or expand a compartment without changing the notation model
- ArrowEffect - Draws an arrow between two EObjects
- AnalysisEffect - Performs graph analysis and shows results in the Analysis view
- FocusContextEffect - Reduce level of detail for context elements, show all details for focus elements (e.g. by collapsing/expanding compartments)
- TransformationEffect - Executes a model to model transformation according to passed information.
- RefreshGMFEditPoliciesEffect - Refreshes the EditPolicies of an GMF editor.
- UnlockEffect - Notify all waiting threads on a given Object. Can be used to synchronize threads with the execution of effects.

Combinations

- LayoutCombination - Performs automatic layout after key combo/layout button is pressed
- RandomLayoutCombination - Perform random layout without changing any layout options.
- KSBasECombination - Used to execute a ksbase transformation.
- LayoutAfterCollapseExpandCombination - Performs layout after a compartment was collapsed or expanded
- ShowHierarchyCombination - Paints the diagram like a rainbow depending on the hierarchy levels
- LayoutAnalysisCombination - Performs automatic analysis after layout
- ManualFocusCombination - Do Focus&Context viewing, where the focus are the selected elements in a diagram. Currently supported for Syncharts and KAOM-Diagrams.
- E2STransformationCombination - Performs a Esterel to Syncharts transformation and following a Syncharts optimization.

Syncharts specific

- LayoutAfterModelChangedCombination - Performs layout after the model was changed
- SignalFlowCombination - Displays the Dual Model
- SyncChartsCombination - Highlights/Collapses/Expands states during simulation
- HighlightSelectedTransitionsCombination - Higlights selected transitions (and all children) in Syncharts. Used to better see which transition belongs to which label.