OSATE Graphical Editor

Developer Guide

2021-08-26

Contents

List of Tables

List of Figures

1 Introduction

This document is intended to provide an overview of the implementation of the Open Source AADL Tool Environment (OSATE) graphical editor. The graphical editor is the component of OSATE which provides diagram-based visualization and editing of AADL models. This document is intended to serve as a starting point for understanding the graphical editor implementation. It is intended as a supplement rather than replacement for the Javadoc documentation.

2 Overview

The graphical editor is composed of Eclipse plugins. These plugins register Eclipse extensions, provide OSGi services, and define extension points that are used to provide support for Business Objects (4.1).

2.1 Plugins

The o*rg.osate.ge* plugin contains most of the graphical editor code. It is the original plugin. As development has progressed, additional plugins have been created as part of efforts to improve the modularity. The graphical editor plugins are shown in Figure 1.

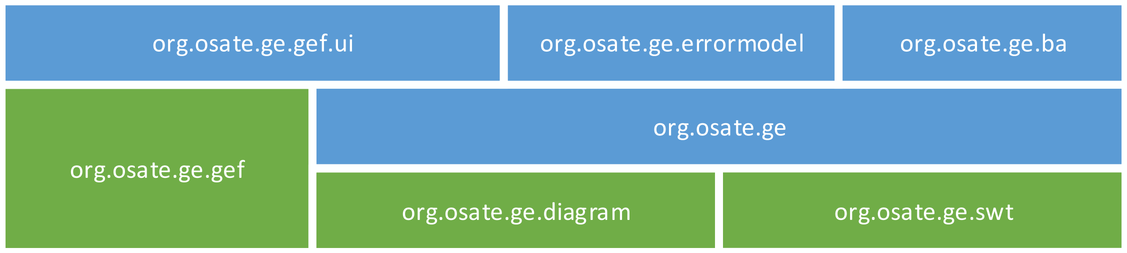

Figure 1. Plugins

The green plugins are those which are not dependent on the Eclipse workbench UI. Plugins may use internal APIs provided by other graphical editor plugins; to avoid compatibility issues, all installed graphical editor plugins must be from the same version of the graphical editor. Plugins such as *org.osate.ge.gef* and *org.osate.ge.swt* contain classes which are related to AADL concepts; however, *org.osate.ge*, *org.osate.ge.errormodel*, and org.osate.ge.ba are the only plugins which depend on the AADL meta-model. Brief descriptions of the plugins are contained in Table 1.

Table 1. Plugins

Name	Description
org.osate.ge	The original graphical editor plugin. Contains the stable API and non JavaFX portions of the diagram editor. It defines services and extension points. It also provides the AADL declarative and instance model support.
org.osate.ge.ba	Behavior annex support. Defines extensions providing behavior annex support in the diagram editor and properties view.
org.osate.ge.diagram	Contains an Ecore model defining the **Error! Reference source not found.** (4.2.2).
org.osate.ge.errormodel	Error model annex support. Defines extensions providing error model annex support in the diagram editor and properties view.

org.osate.ge.gef	Contains JavaFX nodes for graphics and the palette used by the graphical editor.
org.osate.ge.gef.ui	JavaFX diagram editor implementation. Integrates into Eclipse workbench. It also contains an implementation of the DiagramExportService service which allows saving diagrams as images.
org.osate.ge.swt	A collection of SWT user interface components intended to loosely follow the Model-View-ViewModel (MVVM) pattern.

The intention is for the "core" of the graphical editor to be contained in *org.osate.ge* and for it to be independent of the AADL2 meta-model. It is felt that decoupling the editor from the meta-model makes it easier to understand, maintain, and reason about the behavior of the graphical editor. Unfortunately, this intention is not fully realized in the current version. While most usages of the AADL meta-model within *org.osate.ge* are contained within the org.osate.ge.aadl2 package and sub packages are registered with the "core" as extensions, other components within this plugin reference the AADL meta-model. For example: the diagram updating mechanism and serialization provides specialized code for supporting AADL properties. Because of this, it is non-trivial to extract the AADL2 specific portions of the graphical editor into a separate plugin.

2.2 API

The graphical editor attempts to expose a stable API to allow adding support for additional business objects without requiring knowledge of the libraries used to implement the graphical editor. The API avoids exposing the graphical editor's internal data structures and the usage of JavaFX. This API consists of the extension points and the packages exported from *org.osate.ge*. All packages which do not have "internal" in the name are considered part of this API. Other graphical editor plugins export packages, but no effort is made to maintain compatibility between versions of these plugins.

The only known usages of the graphical editor's API is the AGREE simulator and the AGREE graphical editing plugins. At the time of writing this document, the state of these plugins is unknown.

2.3 Services

The graphical editor defines several OSGi declarative service components. The services are available globally via the Eclipse service context. A few are public and are available for use by other plugins. Most of these services are internal and exist to hold state that is shared between instances of the diagram editor. Global services are declared in the plugin's manifest and have a component description in the plugin's "components" folder. Table 2 contains a list of these global OSGi services.

Table 2. OSGI Services

Java Interface	Availability
AadlModificationService	Internal
AadlResourceService	Internal
ClipboardService	Internal
DiagramExportService	Public
DiagramService	Internal
ExtensionRegistryService	Internal
GraphicalEditorService	Public
ModelChangeNotifier	Internal

ReferenceBuilderService	Public
ReferenceService	Internal
SystemInstanceLoadingService	Internal

In addition to these global OSGi services, the graphical editor defines other services which are provided as arguments to methods implemented by extensions.

2.4 Extension Points

The graphical editor exposes several extension points. These extension points are briefly described in Table 3. Additional documentation is contained in each extension point's schema.

Table 3. Extension Points

Name	Purpose
org.osate.ge.tooltips	Add to tooltips shown when hovering over a diagram element.
org.osate.ge.images	Register images used as icons by palette commands.
org.osate.ge.referenceLabelProviders	Register a class used to convert References (4.4) to a form suitable to display to the user. The labels are used in cases where the reference cannot or should not be resolved. Examples: context labels in diagram view and restoring missing diagram elements.
org.osate.ge.referenceResolvers	Register a reference resolver: a class used to retrieve a Business Object (4.1) using a Canonical Reference (4.4.1).
org.osate.ge.businessObjectHandlers	Register a business object handler: a class which determines how a Business Object (4.1) is represented in a diagram.
org.osate.ge.businessObjectProviders	Register a business object handler: a class which make business objects (4.1) available to be shown in a diagram.
org.osate.ge.contentFilters	Register labeled predicates which define a set of business objects (4.1). Depending on the type of content filter, these can be always shown or could be shown or hidden by the user using a menu item.
org.osate.ge.diagramTypes	Registers a diagram type: a class which specifies the content filters used to determine what children are shown for a diagram element. Diagram types also defines the AADL properties which are shown by default.
org.osate.ge.paletteContributors	Add items to the palette.

3 User Interface

The user interface implementation is split between plugins. The diagram editor uses JavaFX and depends on the Eclipse Graphical Editing Framework (GEF). The other parts of the user interface use SWT. Figure 2Figure 1. Plugins and the following subsections contain an overview of the major parts of the user interface and the Java code which contains their implementations.

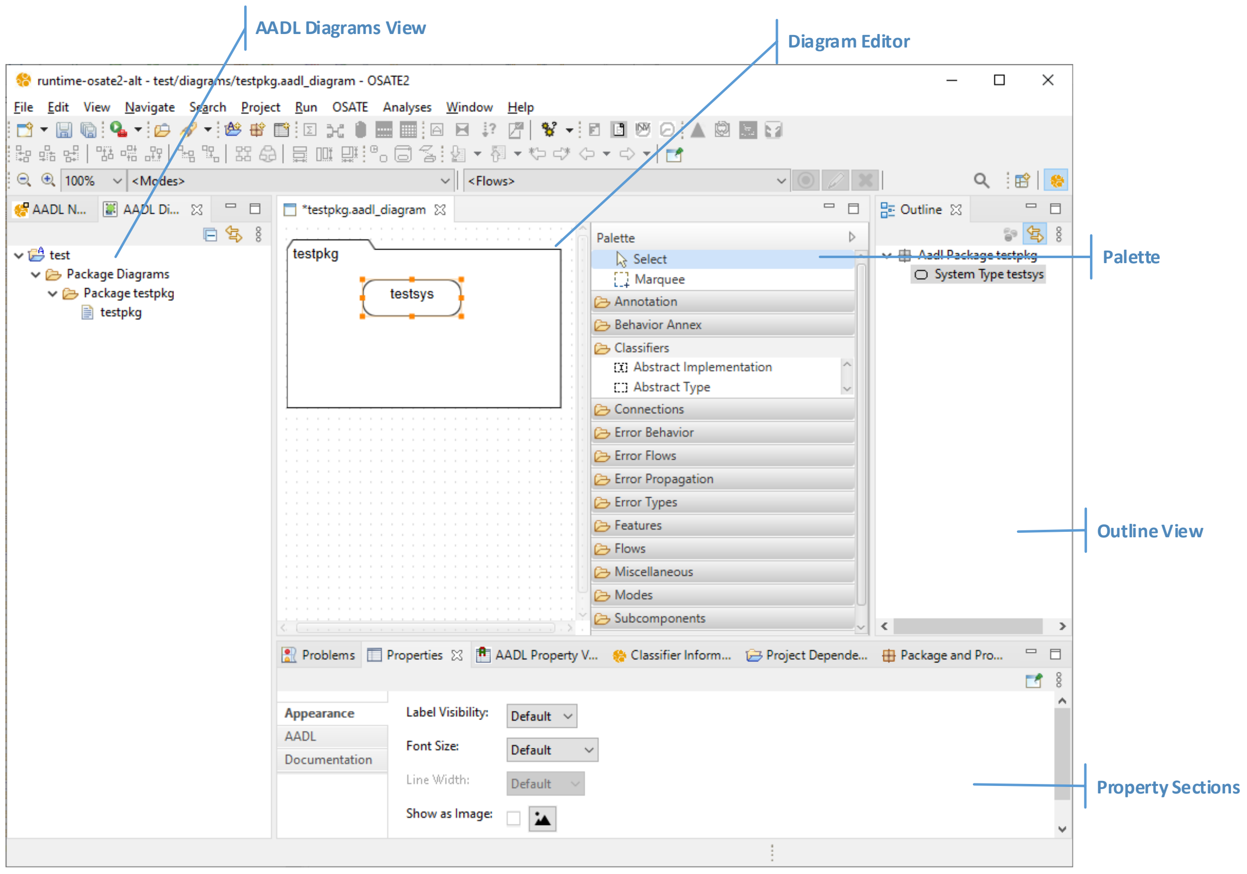

Figure 2. User Interface

3.1 AADL Diagrams View

The AADL Diagrams view shows the diagrams in the workspace. It is implemented in *org.osate.ge.internal.ui.navigator.DiagramsView*.

3.2 Diagram Editor

The diagram editor displays and allows editing diagrams. The editor uses JavaFX and GEF. Its implementation is in *org.osate.ge.gef.ui.editor.AgeEditor*.

3.3 Palette

The palette is a custom JavaFX node (*org.osate.ge.gef.palette.Palette*) which is contained in the editor. The palette model (*org.osate.ge.gef.ui.editor.AgeEditorPaletteModel*) uses the commands provided by registered palette contributors.

The *Select* and *Marquee* tools are defined by the palette model. Additional items are defined by palette contributors. Such contributors implement *org.osate.ge.palette.PaletteContributor* and are registered using the *org.osate.ge.paletteContributors* extension point.

Palette contributors are defined in the following packages:

- *org.osate.ge.palette.internal.AgePaletteContributor* - Notes
- *org.osate.ge.aadl2.ui.internal.palette.AadlPaletteContributor* - Core AADL

- *org.osate.ge.ba.ui.palette.BehaviorAnnexPaletteContributor* - Behavior Annex
- *org.osate.ge.errormodel.ui.palette.ErrorModelPaletteContributor* - Error Model

3.4 Outline View

The outline is implemented in *org.osate.ge.internal.ui.editor.AgeContentOutlinePage* and provided by the editor via *getAdapter()*.

3.5 Property Sections

Property sections are displayed in eclipse's *Properties* view. Property sections are registered using the *org.eclipse.ui.views.properties.tabbed.propertyTabs* and *org.eclipse.ui.views.properties.tabbed.propertySections* extension points.

The property sections are defined in the following packages:

- *org.osate.ge.internal.ui.properties* - Appearance, Documentation, and Note property sections
- *org.osate.ge.aadl2.ui.internal.properties* - AADL
- *org.osate.ge.ba.ui.properties* - Behavior Annex
- *org.osate.ge.errormodel.ui.properties* - Error Model

4 Key Concepts

4.1 Business Object

A business object is an object which is represented by a part of the diagram. AADL model elements such an AADL System Type object are types of business objects. In the code, "bo" is often used as an abbreviation.

4.2 Diagram

A diagram is a tree of diagram nodes. There are two types of nodes: diagrams and diagram elements. A diagram is the root of the tree. Each diagram has a context that restricts which business objects may be represented by child diagram elements. If a context is not defined, the Eclipse project containing the diagram is used as the context. An example of the structure of the contents is shown in Figure 3.

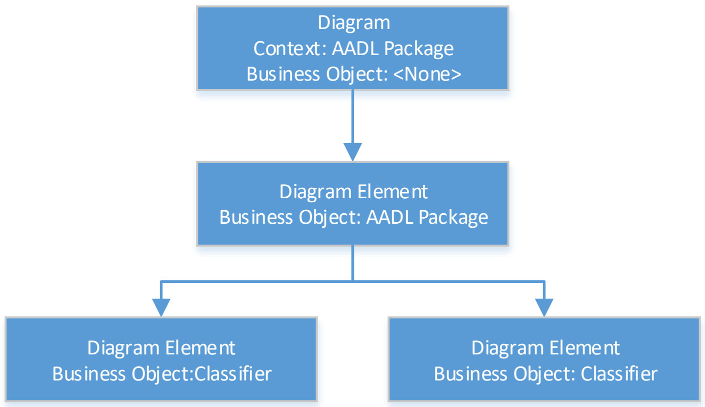

Figure 3. Example Diagram Structure

There are two representations of the diagram: the Ecore based Serialized Diagram which is stored on the disk and the Runtime Diagram which is the data structure used when a diagram is open. The two formats are similar. Although having two separate representations can be beneficial, it can also be a source of confusion. Classes in separate packages often use identical names. Additionally, due to two-stage (create then update) process by which the runtime diagrams are created from the serialized model, it can be difficult to enforce invariants regarding validity of runtime fields. It is worth considering replacing the two representations with a single representation of the diagram to which additional runtime data could be attached. Such an implementation may allow a cleaner implementation without impacting performance.

4.2.1 Diagram Element

All non-root diagram nodes are diagram elements. Each diagram element is associated with a business object. A diagram element has a graphic which determines how it is displayed. A graphic is a shape, connection, or flow indicator. A shape is a rectangle, polygon, label or similar graphic. A connection is an edge between two diagram elements. A flow indicators is an edge which have one moveable end and another end attached to an existing diagram element. In most cases, flow indicators are treated as a type of connection.

4.2.2 Serialized Diagram

The primary purpose of the serialized diagram is to define a stable format for the persistent version of the diagram. The intention of the serialized diagram is to contain the minimal amount of information needed to persist the diagram in a predictable format that can be easily merged with diff and merge tools. The serialized model avoids duplicating information that is in the AADL model. The implementation and the Ecore model is contained in the *org.osate.ge.diagram* plugin. Serialized diagrams contain a format version attribute which contains the file format version. This is used to warn of potential compatibility issues.

4.2.3 Runtime Diagram

A separate runtime diagram was created as a means to improve the performance of the graphical editor. Having a specialized runtime diagram data structure allows flexibility to optimize and refactor the implementation without impacting compatibility with the diagram file format. Each runtime diagram nodes contain a map which allows quickly finding a child using the immutable business object references described in section 4.4. Diagram nodes contains additional fields which cache information contained in the AADL model or provided by the business object handlers. These include such fields as business object, graphics, or an associated diagram element. The implementation of the runtime diagram is contained in the *org.osate.ge.internal.diagram.runtime* package.

The *org.osate.ge.internal.diagram.runtime.DiagramSerialization* class allows serializing and deserializing diagrams. Although that class is used to create the initial runtime diagram from a serialized diagram, it cannot fully populate the runtime diagram. That requires updating the diagram as described in section 5. Because serialized diagrams store minimal information, the AADL model business object providers, and business object handlers are required to fully initialize a runtime diagram.

4.3 Business Object Selection

A business object selection represents a collection of selected business objects. An adapter factory, *org.osate.ge.internal.selection.AgeBusinessObjectSelectionAdapterFactory,* is registered which allows retrieving an instance of *org.osate.ge.BusinessObjectSelection* from an *ISelection* instance. The interface is implemented by *org.osate.ge.ui.UiBusinessObjectSelection*.

This interface allow components of the graphical editor such as property sections to easily edit the business objects associated with the selected diagram elements. For example: a property section can use *BusinessObjectSelection* to allow setting the classifier of all selected subcomponents without being concerned with how the business objects were selected or the packages in which they are contained. It also ensures that the user interface is decoupled from the process by which the AADL model is modified.

4.4 References

References are used to refer to Business Objects. References are created for a business object by the associated business object handler. References are immutable and serializable. There are two types of references which are described in the following sections: canonical references and relative references. Ideally reference should not change when the model changes. In general, references change when elements are renamed. Updating such references are handled by the refactoring process described in section 7. This is not always possible. For example: if a mode transition does not have a name, a reference is build which uses the attributes of the mode transition; admittedly, this may not have been the best approach for this particular type of business object.

4.4.1 Canonical Reference

Canonical references are unique identifiers for a business object. If an appropriate reference resolver extension is registered, canonical references can be resolved to retrieve the referenced business object. Such resolvers are typically only needed for business object types which serve as the context for a diagram. Because changes to the model may invalidate a large number of canonical references, canonical references are only stored to disk when absolutely required. Canonical references are used to persist the diagram's context.

4.4.2 Relative Reference

The business object referenced by a relative reference is dependent on the context in which it is used. For example: the business object referred to by the reference "classifier a" is dependent on the parent package. Usually relative references are relative to a parent diagram element. Relative references are stored in diagram files to associate a diagram element with a business object. The use of relative references reduces the number of references which are invalidated when a model element is renamed. Unlike canonical references, relative references are not "resolved"; they are typically compared with the references of sibling business objects.

4.5 Partial Connections

Some connections are displayed as dotted lines. In the user guide these are referred to as "Abstract Connections". In the source code, these are referred to as "partial" connections. The dotted style indicates that one or more of the true endpoints of the connection is not shown in the diagram. For example: a connection which references a feature contained in a feature group may be shown as a partial connection with the feature group if the feature not being shown in the diagram.

Partial connections are important for usability. Connections must have valid endpoints to exist in the diagram. By showing a connection as a partial connection, the graphical editor avoids removing it completely from the diagram when the endpoint is hidden. Additionally, the user may select to show a connection; if one or more of the end points did not exist, then the connection would not be shown and it would cause confusion for the users. By showing connections are partial connections whenever possible, the cases in which unexpected behaviors occur are reduced.

5 Diagram Update Process

Whenever a model change is detected by the graphical editor, the editor updates the diagram to reflect those changes. The process by which the diagram is updated is shown in Figure 4.

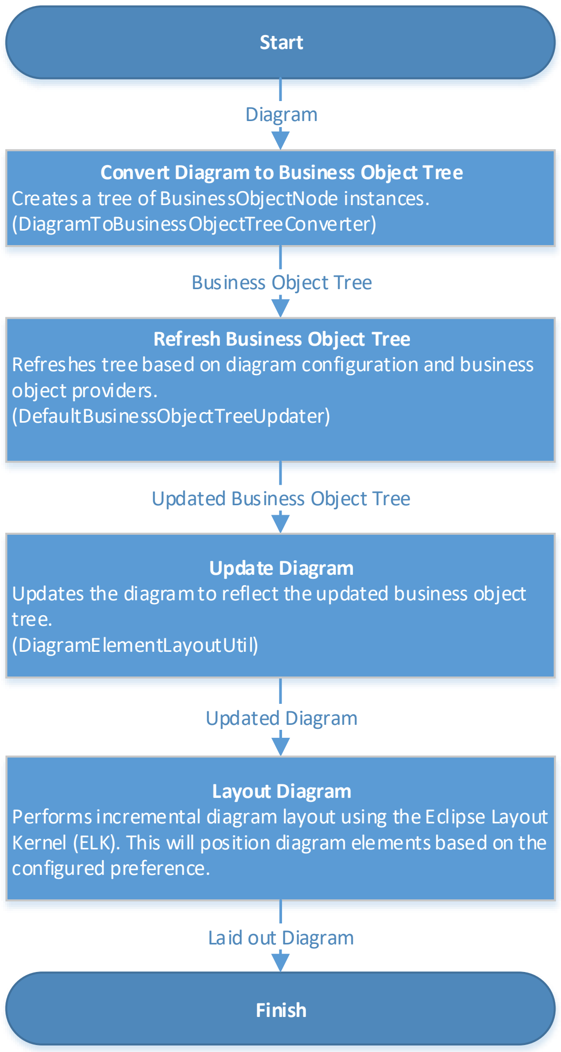
Start
Diagram
Convert Diagram to Business Object Tree
Creates a tree of BusinessObjectNode instances.
(DiagramToBusinessObjectTreeConverter)
Business Object Tree
Refresh Business Object Tree
Refreshes tree based on diagram configuration and business object providers.
(DefaultBusinessObjectTreeUpdater)
Updated Business Object Tree
Update Diagram
Updates the diagram to reflect the updated business object tree.
(DiagramElementLayoutUtil)
Updated Diagram
Layout Diagram
Performs incremental diagram layout using the Eclipse Layout Kernel (ELK). This will position diagram elements based on the configured preference.
Laid out Diagram
Finish

Figure 4. Diagram Update Process

5.1 Convert Diagram to Business Object Tree

First, the diagram is converted to a tree of *BusinessObjectNode* instances. A *BusinessObjectNode* represents the aspects of the structure of the diagram relevant to determining which business objects should be represented in the diagram. A *BusinessObjectNode* is created for each *DiagramNode*. *BusinessObjectNode* instances are also created for business objects which have been recently created by the user using the graphical editor. For example, if a user creates a classifier using the palette then the converter will be called so that a *BusinessObjectNode* is created for the classifier. This will ensure that the classifier is added to the diagram in addition to being added to the model.

5.2 Refresh Business Object Tree

Next, a new business object tree is created based on the converted business object tree and the business objects provided by the registered business object providers. Nodes which reference non-existent business objects are removed. The resulting nodes reference business objects which are not EMF proxies.

Additionally, nodes are created for AADL properties referenced in the diagram configuration. The behavior of AADL properties are different from other business objects. Nodes are created and removed automatically based on the diagram configuration. Handling AADL properties in the *DefaultBusinessObjectTreeUpdater* is a significant source of coupling to the AADL2 model outside of the org.osate.ge.aadl2 package. A more flexible approach would be to have a business object provider which provides business objects for all property values. This would allow the user to hide and show the diagram nodes for individual property values.

5.3 Update Diagram

Next, the diagram is updated to reflect the new business object tree. This step removes and adds diagram elements (4.2.1) to match the structure of the business object tree. If a shape was created using the palette, it uses the position specified during creation to set the position of the new diagram element.

5.4 Layout Diagram

The last step of the process is to perform incremental diagram layout. This process lays out the diagram based on the incremental diagram layout preference. Diagram layout is described in more detail in section 8.

6 Model Modification Process

When the AADL model is modified, open diagram editors are updated to show the modified model. This process is shown in Figure 5.

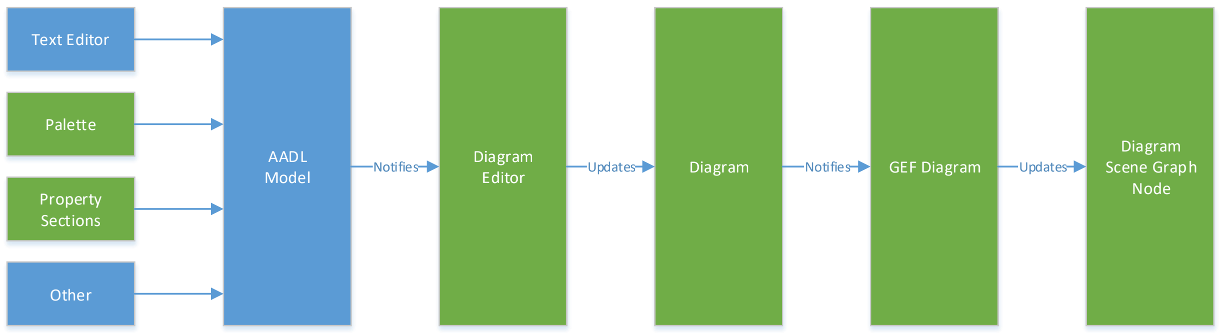

Figure 5. Model Modification Handling

6.1 AADL Model Modifications

The AADL model is modified by the text editor, palette commands, property sections or other mechanism. Modifications made by the graphical editor use the *AadlModificationService* to modify the

AADL model. That service is implemented by *DefaultAadlModificationService* and either modifies an open Xtext document or an EMF resource managed by the graphical editor.

6.2 AADL Model Change Notification

Regardless of how the model is modified, a notification is generated by the *ModelChangeNotifier* service which is implemented by *DefaultModelChangeNotifier*.

6.3 Diagram Updates

When a diagram editor received a model change notification, it updates the diagram as described in section 5.

6.4 Scene Graph Updates

The diagram editor displays the diagram using a JavaFX scene graph. The *GefAgeDiagram* class is responsible for creating the scene graph for a diagram and for updating the scene graph to match the diagram. It can also update the diagram to match the scene graph. This is often useful when implementing interactive diagram modification operations; at the end of the operation, the diagram is modified so that it matches what the user sees.

7 Refactoring

The graphical editor registers a Language Toolkit (LTK) rename participant which updates diagrams when a model is renamed via a refactoring operation. The rename participant updates references in diagram files and open diagrams to reflect the new name. Once references are updated, diagrams are updated using the process described in section 6.

Unless the business object handler implements *CustomRenamer*, the graphical editor renames model elements using an Xtext rename refactoring.

8 Layout

The diagram layout process sets the positions and size of diagram elements. The graphical editor uses the Eclipse Layout Kernel (ELK) to do this. The diagram is converted to an ELK graph, an ELK layout is performed, and the diagram is updated to reflect the resulting layout. In most cases an incremental layout is performed. The behavior of incremental layouts is determined by the incremental layout mode set in the user's preferences. When a diagram contains a feature groups, the process is more complicated and multiple ELK layouts are performed. This is required because nested features cannot be represented in the ELK graph. Diagram layout is implemented in the *org.osate.ge.internal.diagram.runtime.layout* package.

The term layout is also used in the context of JavaFX. In JavaFX, scene graph nodes position and size their children during the layout process. The position and sizes of the diagram elements are used to configure the JavaFX scene graphs. In some cases, diagram elements do not exist for a scene graph node or do not contain positions. In cases such as those, the positions are calculated as part of the JavaFX scene graph node layout. An example of such a case is the positioning of a shape's primary label.

9 Testing

The graphical editor tests are contained in the *org.osate.ge.tests* plugin. The plugin contains both unit and end-to-end tests. Most tests are end-to-end tests which automate the use of the graphical editor using SWTBot. The JavaFX Robot and JavaFX event injection is used in addition to SWTBot to test the JavaFX portions of the editor. In order to try to achieve more comprehensive test coverage, JavaFX event injection is only used in cases where test compatibility issues were encountered.