



DevOps Model RealTime

Model Fixup Plugin

*Author: Mattias Mohlin
Senior Software Architect
IBM*

INTRODUCTION.....	2
INSTALLATION.....	3
VERSIONS.....	3
USING THE MODEL FIXUP PLUGIN.....	5
RECOMMENDATIONS ON THE ORDER OF RUNNING FIXUPS.....	6
MODEL FIXUPS.....	7
ALIEN SYMBOLS REMOVAL.....	7
DUPLICATE IDs REMOVAL FOR TEMPLATE PARAMETERS.....	7
UNNECESSARY REGIONS REMOVAL.....	9
UNNECESSARY ENTRY/EXIT POINTS REMOVAL.....	9
UNNECESSARY JUNCTION POINTS REMOVAL.....	10
OBSOLETE URI PROPERTY REMOVAL.....	11
UNBOUND TYPE REFERENCES RESOLUTION.....	11
DELETE NAME FOR STATE CHART DIAGRAMS CALLED 'STATE DIAGRAM'.....	12
DEPENDENCY FIXUP.....	13
INCOMPLETE CONNECTION VIEWS REMOVAL.....	13
INTERNAL TRANSITIONS INVALID VIEW REMOVAL.....	14
FILE FORMAT FIXUP.....	15
INCORRECT TYPE REFERENCES FIXUP.....	15
OBSOLETE PACKAGE IMPORT REMOVAL.....	16
OBSOLETE PROFILE APPLICATION REMOVAL.....	16
DANGLING FRAGMENT DETECTOR.....	16
CONNECTOR ENDS UPDATER.....	17
SETTING ELEMENTS FOR MULTIPLICITY LABELS ON STRUCTURE DIAGRAMS.....	18
ADD MISSING PACKAGE IMPORTS.....	19
USE CDATA IN MODEL FILES.....	20

This document describes the Model Fixup plugin which is a utility for detecting and fixing problems in a DevOps Model RealTime model which are hard to detect and fix manually.

The document was last updated for Model RealTime 11. All screen shots were captured on the Windows platform.

Introduction

Most problems in an Model RealTime model can be detected and fixed by using standard Model RealTime features such as model validation and by editing the model in various views and diagrams. However, certain problems may be more hidden and may therefore be difficult to detect and fix. Over time several categories of such problems have been identified by IBM and the Model RealTime user community. The Model Fixup plugin provides utilities that allow you to check if your model contains any problems of those known categories. If such problems are found the plugin offers a possibility to correct them.

There are a number of different reasons why a model may need to be corrected using a model fixup:

- The RoseRT importer had (and may still have) limitations and bugs at the time when a model was migrated to Model RealTime. Hence, the resulting Model RealTime model may not always be optimal in all aspects.
- There may have been bugs in older versions of Model RealTime which were used to edit the model in the past. Those bugs may have caused certain parts of the model to be faulty, or in need of improvement.
- New features in later versions of Model RealTime may have caused certain legacy model constructs to become more or less obsolete.
- Users may, for various reasons, have made modeling mistakes that are hard to detect and fix without some kind of tool support. In particular merging model is something that can introduce errors if not used correctly.

The Model Fixup functionality is packaged as a separate plugin rather than being an integrated part of the standard Model RealTime installation. There are two main reasons for this:

- New categories of problems may be found at any time, and by being independent of a certain Model RealTime release, IBM can provide updated versions of the Model Fixup plugin more rapidly.
- It is expected that only a few users within an organization will run the Model Fixup plugin, and most of them will only run it once. Therefore it is not desirable to expose the commands for running model fixups in the general Model RealTime user interface, since it may confuse the majority of users who don't need to run it.

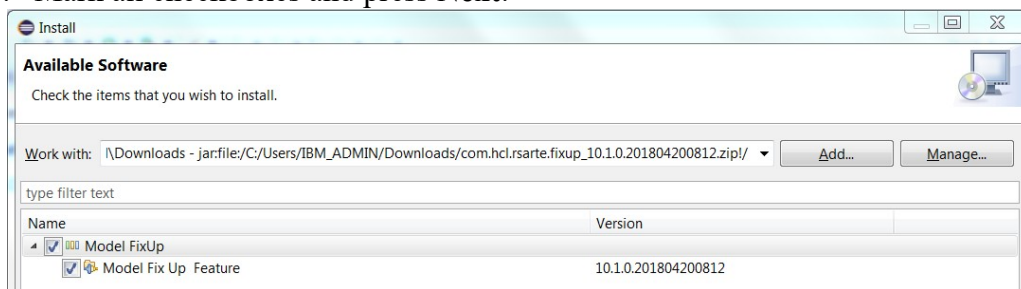
The rest of this document describes how to install the Model Fixup plugin and how to use it for detecting and fixing model problems. Each provided model fixup will then be described in detail.

Installation

Older versions of the Model Fixup utility consists of a single file called `com.ibm.xtools-s.umltdt.fixup_<version>.<date and time>.jar`. Install it by placing this file in the `dropins` folder of the Model RealTime installation. Then start Model RealTime once with the `-clean` option set in `eclipse.ini` to make sure that Model RealTime recognizes and loads the new plugin.

Starting from version 10.1.0.201804200812 (the April 2018 version) the Model Fixup is instead delivered as a P2 update site. This means that you can install it on top of Model RealTime using the same method as when installing Model RealTime itself on top of Eclipse.

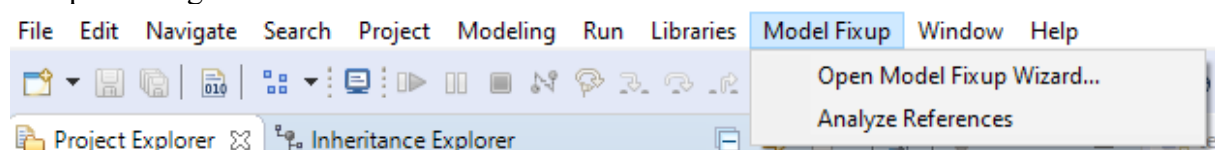
1. Perform **Help - Install New Software**
2. Press the **Add** button and give a unique name to the P2 repository (e.g. `ModelFixup_<today's date>`). Then press the **Archive** button and browse to the location of the P2 update size ZIP file. Press the **OK** button.
3. Mark all checkboxes and press **Next**.



4. Press **Finish** to start the installation. Once it is completed you will be prompted to restart Model RealTime. Do so.

Note: To install in Model RealTime 11.1 it's important to use Java 11. See the Model RealTime 11.1 installation instructions for more information.

If the Model Fixup utility was successfully installed you should see a new menu “Model Fixup” in the global toolbar:



Versions

The table below summarizes which version of the Model Fixup plugin that should be used with which version of Model RealTime / Model RealTime:

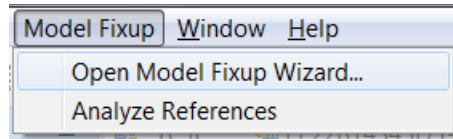
Model Fixup Version	Install Artifact	Model RealTime / Model RealTime version
1.0	Plugin	Model RealTime 8.5.1 (with CP1 or later installed)
2.0	Plugin	Same as 1.0 and also supports Model RealTime 9.0.0.1

3.0	Plugin	Model RealTime 9.0.0.1
4.0 and 5.0	Plugin	Model RealTime 9.1.2
5.2	Plugin	Model RealTime 10.0
10.1	P2 update site	Model RealTime 10.1
10.2	P2 update site	Model RealTime 10.2
10.3	P2 update site	Model RealTime / Model RealTime 10.3
11.0	P2 update site	Model RealTime / Model RealTime 11.0 or later

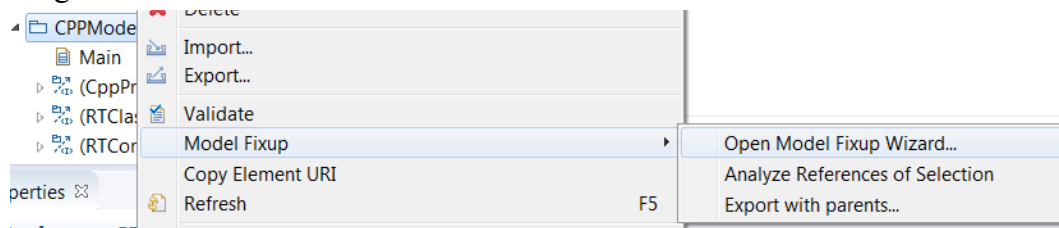
Using the Model Fixup Plugin

Model fixups are run using a Model Fixup wizard. There are two ways to launch this wizard:

- From the global **Model Fixup** menu.
In this case the fixup will be run on all model files in the workspace.



- From the Project Explorer context menu of one or many selected elements.
In this case the fixup will be run only on the model files to which the selected elements belong.



Note that some fixups do not operate on model files, but on other files that are used together with the model, such as transformation configuration files. If you want to run such a fixup you should therefore invoke the Model Fixup wizard either from the global **Model Fixup** menu, or from the context menu on a Project Explorer selection from where the relevant files can be obtained (for example a project).

It is recommended to first try a model fixup on a small part of your model and study its impact before taking the decision to run it on all your models.

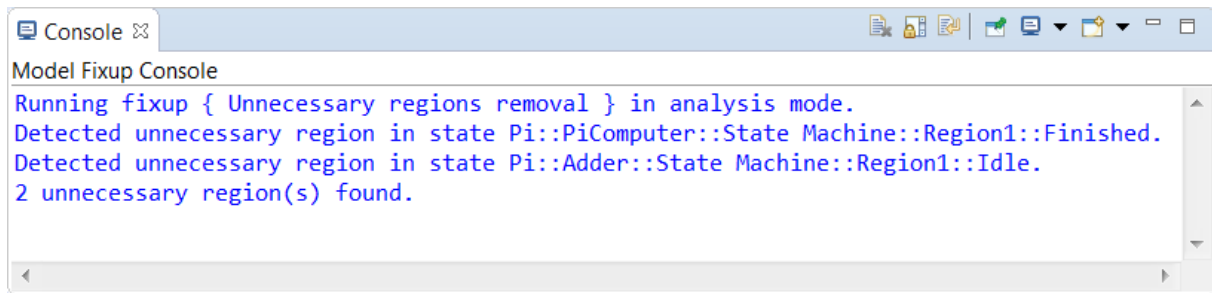
The first page of the Model Fixup wizard presents a list of available fixups and lets you choose which one to run. Each fixup in the list is capable of detecting and fixing one particular category of problems. A brief description of the selected fixup is shown in the text area to the right. The fixups are described in more detail later on in this document.

Each fixup can be run in two different modes:

- Analyze model and report found problems**
In this mode the selected fixup will only analyze the model and report problems that are found. The models will not be modified. This is the default mode.
- Update model and report fixed problems**
In this mode the selected fixup will both find problems and fix them.

It is always recommended to first run a fixup in analysis mode to see if your model contains any problems which the selected fixup can detect and fix. Note that a few of the fixups do not support the analysis mode and can only update the model directly.

Information about found and fixed problems are printed to a special **Model Fixup** console:



```
Model Fixup Console
Running fixup { Unnecessary regions removal } in analysis mode.
Detected unnecessary region in state Pi::PiComputer::State Machine::Region1::Finished.
Detected unnecessary region in state Pi::Adder::State Machine::Region1::Idle.
2 unnecessary region(s) found.
```

Problems are also reported in the Problems view, and from there you can navigate to the place in the model where the problems were found by double-clicking on them. You may mark the checkbox **Clean all problem markers before running model fixup** in order to ensure that the Problems view get cleared before the fixup starts to run.

When you are ready to run a model fixup in “Update” mode you should keep the following in mind:

- Make sure you treat the running of a model fixup as a refactoring operation. This means that you should check-in all changes made by a model fixup as a whole, and not mix them with other unrelated changes. Also you should check-in the changes made by one model fixup before running another model fixup. Remember that changes made by a model fixup are not undoable, so it is important to be able to revert back to a good state in case you decide to rollback a model fixup.
- Model fixups may yield a huge number of changes in a model, which can lead to many changes and conflicts in Compare/Merge. Make sure you don't get any conflicts when checking in your models after a model fixup has been run.
- Usually you can choose to run fixups in any order, but some fixups are recommended to be run before others. If you decide to run multiple fixups you should run them in the same order as they are listed in the Model Fixup wizard (see also [Recommendations on the Order of Running Fixups](#)).
- Some model fixups may have side-effects such as closing all open files in the workspace, to be able to carry out model changes in a safe way.

There is a checkbox **Save model files after running model fixup** that can be set for those fixups which do not already save changed files automatically. Note that if you don't set this option, you must manually save changed model files after running the fixup. It may be necessary to make a selection in the Project Explorer to enable the **Save** and **Save All** buttons in the toolbar.

Recommendations on the Order of Running Fixups

Even if each model fixup can be run independently, it is recommended to run the following fixups in the described order to avoid reprocessing of certain model elements that have been already cleaned-up:

- Unnecessary regions removal
- Unnecessary entry/exit points removal
- Unnecessary junction points removal

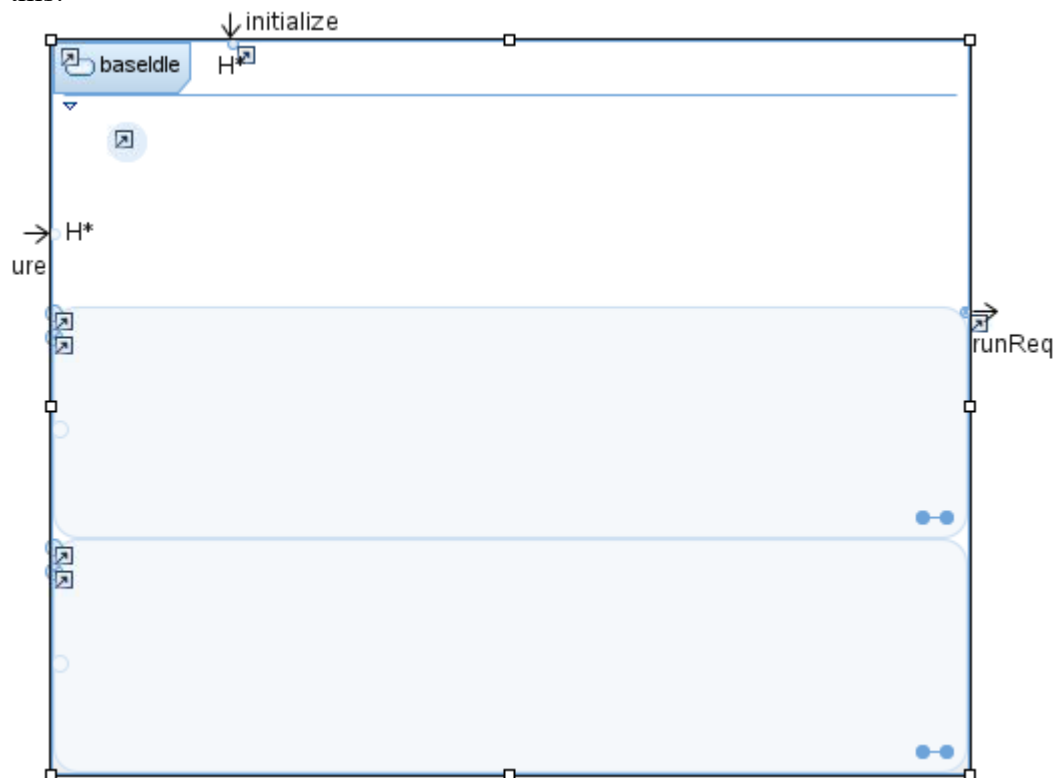
In general the recommendation is to run fixups in the order in which they are listed in the Model Fixup wizard, even if the order of execution only is significant for a few of the available fixups.

Model Fixups

All model fixups that are provided by the Model Fixup plugin are described below.

Alien symbols removal

An alien symbol is a symbol which is not connected to a semantic element in the model, but where such a semantic element has to exist for the model to be well-formed. A model that contains alien symbols is corrupted, and should be corrected by removing those symbols. Alien symbols may lead to many different symptoms. For example, you may get errors when opening a diagram. A specific symptom for alien state symbols could be a composite state symbol that has nested state symbols that cannot be selected. It may for example look like this:



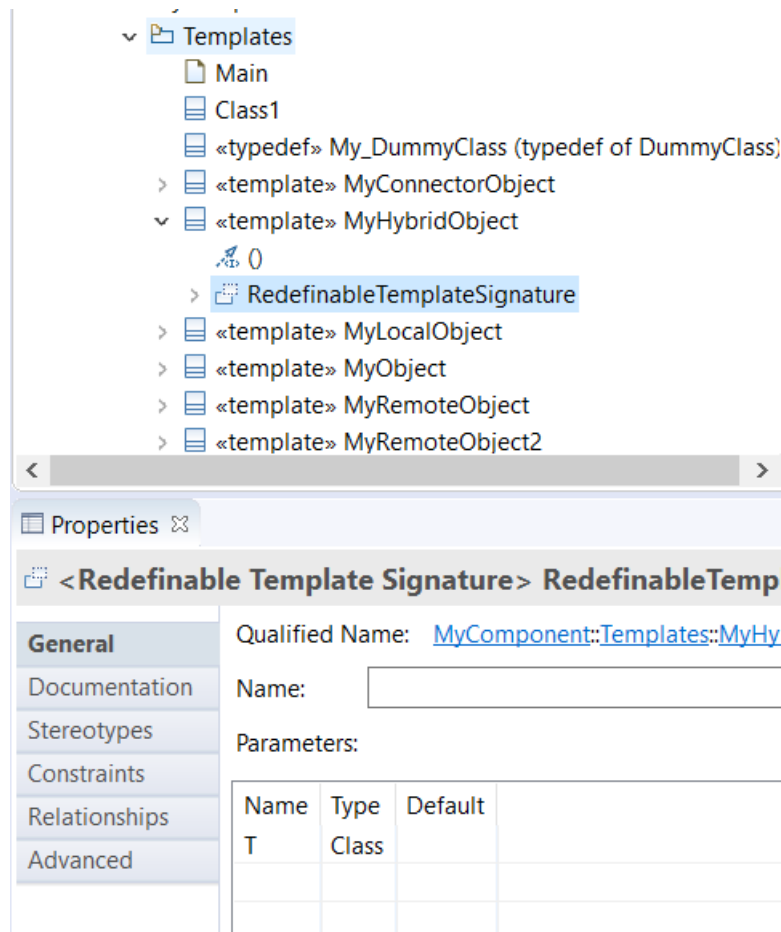
The reason for this corruption is not fully known, but it may originate from merging models using older versions of Model RealTime.

In analysis mode this fixup will report a problem for each alien symbol that is found. In update mode the fixup will correct the model by deleting each found alien symbol.

Duplicate IDs removal for template parameters

Starting with Model RealTime 10.3, all build operations are handled by the model compiler. The model compiler runs in its own process, outside of Model RealTime, and a consequence of this is that the way it loads models is slightly different compared to when loading the models inside the Model RealTime IDE. For example, when models are loaded in the model compiler all presentation elements (diagrams, symbols, lines etc) are skipped since these elements do not affect how code is generated. The model compiler is also more strict, and some minor

errors in the model which the Model RealTime IDE doesn't care about, may lead to errors or faulty generated code when loaded by the model compiler. One example is related to templates. If your model contains a "RedefinableTemplateSignature" with a parameter list that contains duplicate IDs, then the model compiler will treat this as an error that causes the corresponding root element not to be loaded. This, in turn, typically leads to incorrectly generated code. However, the Model RealTime IDE can load such a model without any errors:



Here is an example of the model representation where the duplicate IDs can be seen (the text marked in boldface):

```
<packagedElement xmi:type="uml:Class" xmi:id="_gRW4wF9pEem0Ebcj45TfPQ" name="MyHybridObject"
clientDependency=" iXEoFV9pEem0Ebcj45TfPQ iXFPIF9pEem0Ebcj45TfPQ iXF2MF9pEem0Ebcj45TfPQ
_gRW4119pEem0Ebcj45TfPQ iXEBAl9pEem0Ebcj45TfPQ iXEoEF9pEem0Ebcj45TfPQ iXF2MV9pEem0Ebcj45TfPQ">
  <eAnnotations xmi:id="_gRW4wV9pEem0Ebcj45TfPQ" source="http://www.eclipse.org/uml2/2.0.0/UML">
    <details xmi:id="_gRW4w19pEem0Ebcj45TfPQ" key="template"/>
  </eAnnotations>
  <ownedTemplateSignature xmi:type="uml:RedefinableTemplateSignature"
xmi:id="_gRW4xF9pEem0Ebcj45TfPQ" parameter="_gRW4xV9pEem0Ebcj45TfPQ _gRW4xV9pEem0Ebcj45TfPQ
_gRW4xV9pEem0Ebcj45TfPQ _gRW4xV9pEem0Ebcj45TfPQ">
    <ownedParameter xmi:type="uml:ClassifierTemplateParameter" xmi:id="_gRW4xV9pEem0Ebcj45TfPQ"
parameteredElement="_gRW4x19pEem0Ebcj45TfPQ">
      <ownedParameteredElement xmi:type="uml:Class" xmi:id="_gRW4x19pEem0Ebcj45TfPQ" name="T"
templateParameter="_gRW4xV9pEem0Ebcj45TfPQ"/>
    </ownedParameter>
  </ownedTemplateSignature>
  <interfaceRealization xmi:id="_gRW4119pEem0Ebcj45TfPQ" client="_gRW4wF9pEem0Ebcj45TfPQ">
    <supplier xmi:type="uml:Interface"
href="platform:/resource/SELAdaption/SelAdapterFramework.efx#_kKc4sM-eEeeh20cXHI3AQQ"/>
    <contract href="platform:/resource/SELAdaption/SelAdapterFramework.efx#_kKc4sM-
eEeeh20cXHI3AQQ"/>
  </interfaceRealization>
</packagedElement>
```

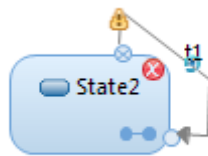
The cause of this corruption is currently unknown. It might be related to merge operations.

Unnecessary regions removal

Older versions of Model RealTime had a bug in the condition for when to show the "composite" decorator on a state (••). Another bug was that when you double-clicked on a non-composite state, but then answered No in the dialog that asks if a sub statemachine should be created, then the state still became composite (although not shown to be composite). And this modification to the model was not undoable. A consequence of these bugs is that many models contain composite states that were never intended to be composite. If such composite states are double-clicked you will just find an empty state chart diagram.

This fixup detects composite states that contain empty regions, or regions that only contain an initial pseudo-state that is not redefined in inherited state machines. By removing such unnecessary regions the composite state decorator can be trusted so that only states that are true composite states show this decorator. Another benefit with this clean-up is that the model becomes smaller without those unnecessary regions.

Here is an example of a composite state where the region is empty:



It is possible to remove the region for an individual state by means of a Quick Fix that is available for the problems that are reported by the fixup. It can also be done by means of the command **Make Non-Composite** that is available in the context menu of the state. To remove all empty regions run the fixup in "Update" mode.

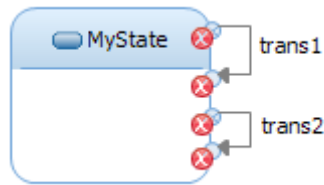
When you run this fixup make sure that your workspace contains all relevant projects which may contain inherited state machines with redefined initial transitions. Otherwise there is a risk that this fixup will think a region is unnecessary even if it is not, and hence remove it.

Unnecessary entry/exit points removal

In older versions of Model RealTime entry/exit points were always created on a state when connecting a transition to it. However, entry/exit points are only useful to have on a composite state. The Rose RT importer also has this behavior which means that if your model was migrated from Rose RT it is likely to contain lots of unnecessary entry/exit points.

This fixup will detect and remove entry/exit points that are attached to non-composite states, and that hence are unnecessary. The benefit with this clean-up is less clutter in state machine diagrams, smaller model files and diagrams that open faster. Editing of transitions that connect directly to a state is also easier compared to if they connect to entry/exit points. In particular it is possible to easily move self-transitions on the state border using arrow keys if they do not connect to entry/exit points.

Here is an example of unnecessary entry/exit points that have been detected by this fixup:

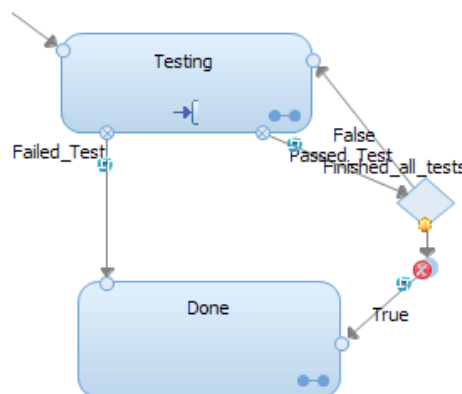


It is possible to remove individual entry/exit points by means of a Quick Fix that is available for the problems that are reported by the fixup. To remove all unnecessary entry/exit points run the fixup in “Update” mode.

Unnecessary junction points removal

The model representation for choice points is different in Model RealTime compared to Rose RT. In Rose RT the choice point condition is associated with the choice point itself, while in Model RealTime the condition is a guard on each transition that go out from the choice point. This difference means that certain Rose RT models where inherited state machines redefine such transitions cannot be directly mapped to Model RealTime models. Because of that, the Rose RT importer migrates a Rose RT choice point using an extra junction point in Model RealTime. The transition between the choice point and the junction point contains the guard condition while the transition that leaves the junction point contains the effect code (if any). This mapping is necessary in order to support inherited state machines where the transition with the effect code is redefined. However, if there are no redefinitions for the outgoing transitions such junction points are superfluous and their incoming and outgoing transitions could be merged without affecting the behavior of the model.

This fixup detects junction points that are unnecessary and could be removed. Here is an example of a junction point in a model migrated from Rose RT which is unnecessary and that could be removed without affecting the behavior of the model:



It is possible to remove individual junction points by means of a Quick Fix that is available for the problems that are reported by the fixup. To remove all unnecessary junction points run the fixup in “Update” mode.

When you run this fixup make sure that your workspace contains all relevant projects which may contain redefined junction point transitions.

Obsolete URI property removal

This fixup operates on transformation configuration files in XML format. It should not be used if your transformation configurations uses the new JavaScript format (.tcjs).

A transformation configuration has a property called “URI” which holds the URI from which the transformation configuration was loaded. This property is only needed in the in-memory representation of a transformation configuration and should not be present in a .tc file. However, due to a bug in an old version of Model RealTime it is possible that the URI property has been incorrectly persisted so that it appears in a .tc file. If such a .tc file has been copied in order to create another transformation configuration the “URI” property will therefore contain the URI of the original transformation configuration which is incorrect. This could lead to that some operations that deal with transformation configurations fail, for example navigation from model elements to generated code.

In analysis mode this fixup scans transformation configuration files for the “URI” property. A typical report may look like this:

```
Running fixup { Obsolete URI property removal } in analysis mode.
Detected obsolete URI property in transformation configuration \
DoxygenTest\HelloWorld.tc.
Detected obsolete URI property in transformation configuration \
DoxygenTest\ExtLib.tc.
Detected obsolete URI property in transformation configuration \
OrganizeSources\HelloWorld.tc.
Detected obsolete URI property in transformation configuration \Sample\
Release.tc.
Detected obsolete URI property in transformation configuration \Sample\
Debug.tc.
5 obsolete properties found.
```

In update mode the fixup will correct the faulty transformation configurations by simply removing the “URI” property.

Note that contrary to most other fixups this fixup operates on transformation configuration files rather than model files. Hence, if you want to run this fixup you should invoke the Model Fixup wizard on something that contains .tc files (the whole workspace, a project, an individual .tc file etc.).

Unbound type references resolution

Models imported from Rose RT often have typed elements (attributes, parameters etc) where the type reference is unbound, and the type is only specified by means of the “Native Type” property. Such models can be correctly transformed to source code, since the “Native Type” property stores the name of the target type. However, many features in Model RealTime will not work correctly for such an element due to the unbound type reference. One symptom of this problem shows up in the Properties view where the **Open Type** button will show that the type reference is unbound.

Type:

Examples of other features which won't work when the type reference is unbound include **Analyze model and add missing dependencies** and **Detect source dependencies automatically**.

Before you run this fixup ensure that your workspace contains all relevant projects so that the correct target types can be located. You want the fixup to find and fix type references that are truly unbound, and not just those that are unbound because the project that contains the target type is currently not present in the workspace.

When the fixup is run in analysis mode a typical report may look like this:

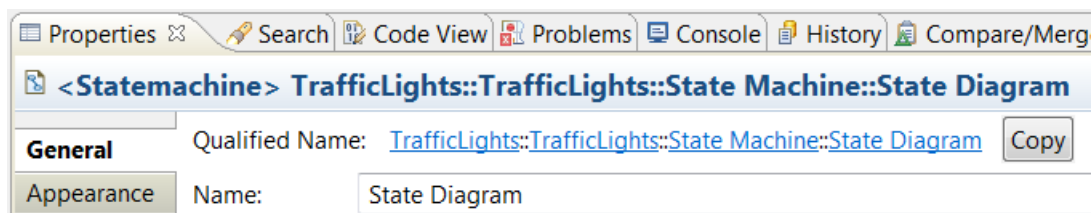
```
Running fixup { Unbound type references resolution } in analysis mode.  
Detected unbound type reference in element TrafficLights::TrafficLights::arr.  
1 unbound type reference(s) found.
```

In update mode the fixup will attempt to resolve unbound type references by searching the workspace for a type that matches what is specified in the “Native Type” property.

Delete name for state chart diagrams called 'State Diagram'

The name of a newly created state chart diagram is by default empty, and then the qualified name of the state machine is used to describe the diagram. Unfortunately the Rose RT importer used to give the fixed name “State Diagram” to all imported state chart diagrams. Thereby they become impossible to distinguish when working in Model RealTime. For example, if you open two different state chart diagrams, the title of both editors is “State Diagram”.

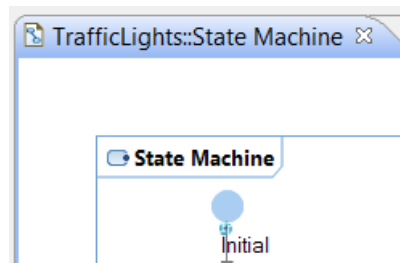
Here is an example:



This fixup resets the name of such diagrams.

```
Applying fixup { Delete name for state chart diagrams called 'State Diagram' }.  
Detected state machine diagram with the name 'State Diagram' in element  
TrafficLights::TrafficLights::State Machine.  
Clearing name of the diagram TrafficLights::TrafficLights::State Machine::State  
Diagram.  
1 diagram name(s) has been cleared.
```

When the diagram name is empty, the qualified name of the state machine is shown in the editor title, which makes it easier to work with many open diagrams at the same time:



Dependency fixup

This fixup performs several consistency checks for dependencies. For example, it has been observed in legacy models that some dependencies have more than one client or supplier. This is usually not noticeable when working with the model in Model RealTime, but can be seen if looking in the model files. The cause of this duplication is unknown, but may likely be related to model merge.

Another problem which this fixup can detect and fix is when a dependency references a client element, but where that client element does not contain the corresponding reference to the dependency. Such a dependency is broken and can cause various problems, such as incorrectly generated C++ code by the model compiler. The problem is difficult to detect and fix using the regular Model RealTime user interface since such dependencies do not show up in the Dependencies tab in the Properties view.

This problem has been observed when the client element is stored in its own fragment file, and it's likely that it was caused by an incorrectly performed file-by-file merge which caused the dependency reference to be lost. The fixup will report and remove all broken dependencies it finds.

Other problems that this fixup can detect include dependencies with unresolved client or supplier references, duplicated dependencies between the same elements (this could affect code generation negatively) and dependencies with no clients or suppliers at all. Some of the problems found by this fixup cannot be automatically corrected. In those cases you instead have to navigate from the Problems view to the location in the .emx or .efx file where the problem was found. You then need to decide if the problem can be fixed there, or if it's easier to remove the entire dependency and recreate it instead.

Note that depending on what problems that are found and fixed, you may need to run this fixup multiple times before all dependency problems have been detected and fixed. When the fixup no longer finds any problems you know that you are done.

Also note that the [File format fixup](#) is explicitly applied on modified model files that are updated by this fixup.

By running this fixup your model files can become smaller, and you prevent future problems which corrupted dependencies possibly could cause.

Incomplete connection views removal

The symptom of this problem is a model validation error:

“Relationships must connect to correct semantic elements”

However, when trying to navigate to the source of this problem you would arrive at a diagram view (i.e. a line) which is incomplete, and that is hard to make complete and correct again. The background for this problem is a bug in the Rose RT importer which led to incomplete views for inherited transitions and connectors.

When running this fixup the diagram views become corrected, so that the model validation error no longer is reported, and that navigation to the diagram views work correctly again. Also, the size of the model becomes a little smaller.

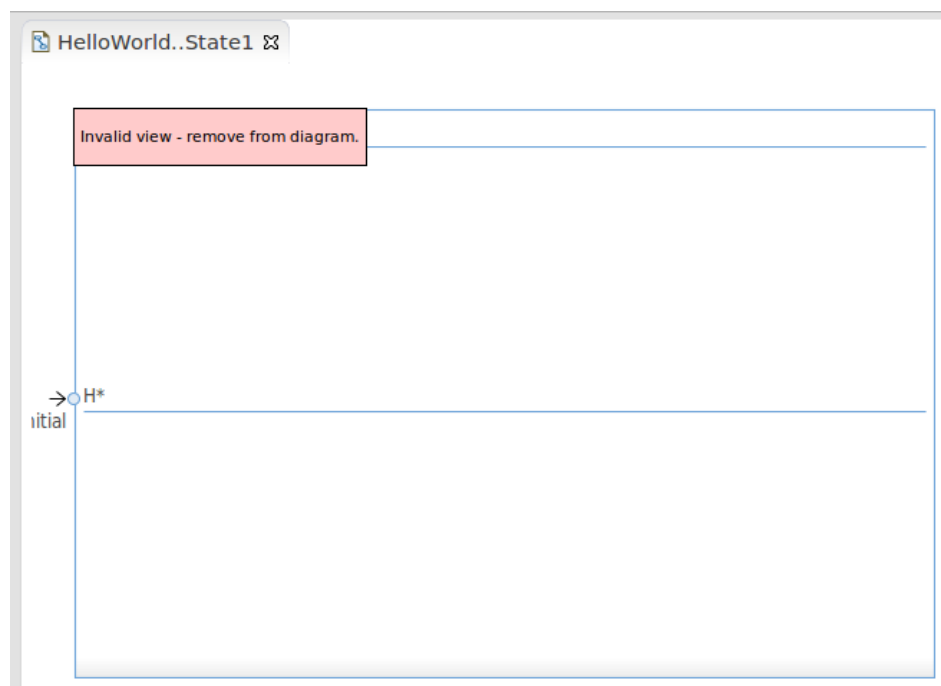
InternalTransitions invalid view removal

The version Model RealTime 9.1.2 CP1 2015.46 introduced an experimental feature to display internal transitions in a separate compartment outside the State Machine diagram. But unfortunately there was a bug in this feature that led to the creation of new view elements when opening an enclosed State Machine diagram. This happened even when the experimental feature was disabled. As a consequence, you might observe 'Invalid view - remove from diagram' elements when the diagram is opened in later versions of the tool where the internal transitions compartment support was redesigned.

This is a fragment of a model file with the element causing the 'Invalid view' message:

```
<contents xmi:type="umlnotation:UMLDiagram"
  xmi:id="_QHleQA0cEeaN8P3y4nEU0Q" type="Statechart" ...>
...
<children xmi:type="umlnotation:UMLFrame"
  xmi:id="_QHleRg0cEeaN8P3y4nEU0Q" type="InternalTransitions" ...>
...
</children>
```

This is how the problem looks like on a State Machine diagram:



This fixup finds and removes all view elements created by that outdated implementation.

File format fixup

It has been observed that when using the [Alien symbols removal](#) fixup in version 3.0 and older of the Model Fixup plugin, the file format of model files is changed in a way that differs from what Model RealTime is used to work with. It has the effect that a textual diff of model files leads to irrelevant changes, and Model RealTime would undo changes the next time the file is saved. The following file format changes have been discovered:

- every self-closing tag is replaced with an explicit end tag: For example,
`<specification xmi:type="uml:LiteralInteger" xmi:id="_1mrEcNWKEeGu9thmUm_ypQ" value="10"/>`
is changed to
`<specification xmi:type="uml:LiteralInteger" xmi:id="_1mrEcNWKEeGu9thmUm_ypQ" value="10"></specification>`
- `"` and `>` is replaced with `"` and `>` respectively
- whitespaces and the order of XML headers may be different
- the tool version stored in the beginning of the file may be changed to the current version, for example:
`<!--IBM Rational Software Architect RealTime Edition 10.2.0-->` changed to
`<!--IBM Rational Software Architect RealTime Edition 10.3.0-->`.

There could also be other situations which may lead to file format changes, such as if manually editing or merging model files.

The implementation of the [Alien symbols removal](#) fixup has now been improved to avoid these unwanted side-effects. If you used the earlier version of this fixup it's now recommended to run the "File format fixup" to normalize the model file format. Note, since "File format fixup" will re-save all files, it may lead to a large amount of changes that can't be tracked using model compare but with text compare.

Incorrect type references fixup

Starting with Model RealTime 10.0 some components became optional to install (e.g. TargetRTS, Connexis, EGit integration, Sketcher and CORBA support). Some other components were completely removed from the product (e.g. RTJava and UAL support).

In Model RealTime 11.0 2020.22 the support for C became an optional install component, and the "C++ Types" package was removed from the product (since it was obsolete and no longer used). In the same version the type "unsigned" in the CppPrimitiveDatatypes was removed since it was a duplicate of "unsigned int".

Some of these components that no longer are available contained datatypes, and it's possible that your models by mistake contain references to some of those datatypes. For example, you may have wanted to set the type of an attribute to the C++ type int, but by mistake picked the Java or UAL int type instead. For example:

```
<type xmi:type="uml:PrimitiveType" href="pathmap://UAL_LIBRARIES/UALPrimitiveDatatypes.emx#_h28iAFIzEd2ex-LDlzlaiIA?UALPrimitiveDatatypes/int?"/>
```

These broken type references may have similar symptoms as described in [Unbound type references resolution](#). However, that fixup cannot solve these problems since the target types no longer exist.

This fixup parses files on XML level and replaces references to types from the packages JavaPrimitiveTypes, RoseCppTypeDatatypes, CORBAPrimitiveTypes, UALPrimitiveDatatypes and "C++ Types" to corresponding types from the CppPrimitiveDatatypes package. The matching is based on type name, and if a matching type is not found in the CppPrimitiveDatatypes package, a message is printed in the console to let you manually correct the type reference. The fixup also updates references to CppPrimitiveDatatypes::unsigned to CppPrimitiveDatatypes::unsigned int.

This fixup has an additional wizard page where you can specify if you want it to also replace references to C types so they instead reference C++ types in CppPrimitiveDatatypes. Mark this checkbox if you don't have the C support installed in Model RealTime.

Note: This fixup does not support the "Analyze" mode.

Obsolete package import removal

This fixup is somewhat related to the "Incorrect type references fixup" and should typically be run after running that fixup. The fixup will remove package imports of the "C++ Types" package, and also for the CPrimitiveDatatypes package if you mark the checkbox on the second wizard page.

Although it's easy to manually remove obsolete package imports in the Model RealTime UI, this fixup can be convenient if you have a large number of models that contain these package imports.

```
Running fixup { Obsolete package import removal } in analysis mode.  
Obsolete Package Import - 'C++ Types' for resource 'CPPModel' with URI  
'platform:/resource/ANY/CPPModel.emx#_wmlcQIMYEegePZtsqr4oGQ'  
1 resource(s) with obsolete package imports found
```

Obsolete profile application removal

This fixup is somewhat related to the "Incorrect type references fixup" and "Obsolete package import removal" and should typically be run after running these fixups. The fixup will remove applied profiles that are obsolete or that were applied by mistake in the past. The "C++ Transformation" profile is an obsolete profile and the "CPropertySets" profile should not be applied unless you have installed the support for C. The latter will therefore only be removed if you mark the checkbox on the second wizard page.

Although it's easy to manually remove obsolete profile applications in the Model RealTime UI, this fixup can be convenient if you have a large number of models that have any of these profiles applied.

Dangling fragment detector

By dangling fragment we mean a model fragment file (.efx) that has become detached from its parent model files (.efx or .emx), but is still present in the workspace. The reason of such corruption is unknown but may possibly be related to compare/merge with ClearCase. There are several types of such fragments:

- a fragment which does not have a reference to its parent, but there is another fragment in the workspace having a "child" reference to it. In this case, the reference to the parent file can be restored and the fragment file hence be repaired.

- a fragment which does have a reference to its parent, but the parent does not have it in the list of its children. In this case, the fragment can be removed from the workspace if there are no other files with references to this fragment.
- a fragment which does not have a reference to its parent, and there are no fragments which both have it as a child or contain references to it. In this case, the fragment file can be removed.

This fixup detects dangling fragments of such types and tries to either repair or remove them. In case none of this can be done, it prints the name of files which have references to a found dangling fragment. This fixup supports the “Analyze” mode, and you can use it to check if your workspace contains any dangling fragments. If so, you can apply a Quick Fix on each found problem in the Problems view. You can also navigate to the dangling fragment in the Project Explorer.

Output examples:

```
Applying fixup { Dangling fragment detector }.
```

```
Detected 4 dangling fragments.
Fixed container reference for C:\RTModels\.models\Uni_Package.efx
Fixed container reference for C:\RTModels\.models\Test_Package.efx
Fixed container reference for C:\RTModels\.models\Resources_Package.efx
Fixed container reference for C:\RTModels\.models\GenericDataClasses_Package.efx
Fixed 4 of dangling fragments
```

```
Applying fixup { Dangling fragment detector }.
```

```
Detected 1 dangling fragments.
Fixed container reference for C:\RTModels\model\nControl.efx
Fixed 1 of dangling fragments
```

Connector ends updater

Although composite structure diagrams are usually placed in capsules, it is also possible to have such diagrams in collaborations. If such a model was migrated from Rose RT, the connectors would not be correctly set-up in Model RealTime. They look correct in diagrams, but in the Properties view one can see that the "Parts" information is missing. Here is an example:

	Connector End	Connector End
Ports	p2	p2
Parts	capsule4 2	capsule4
Protocols	p2	p2

Correct connector in capsule

	Connector End	Connector End
Ports	port1	portA~
Protocols	Protocol1	Protocol1

Incorrect connector in collaboration

More specifically, the reason for the problem is that the Rose RT importer does not set-up the "Part with Port" property of the connector ends in this particular case.

This fixup can detect this kind of corrupted connectors and fix them.

The output in the console may look like this:

```
Applying fixup { Connector ends updater }.
Detected connector end with missing capsule part information in the connector
CPPModel::Collaboration1::Protocol1, Protocol1~.
Detected connector end with missing capsule part information in the connector
```

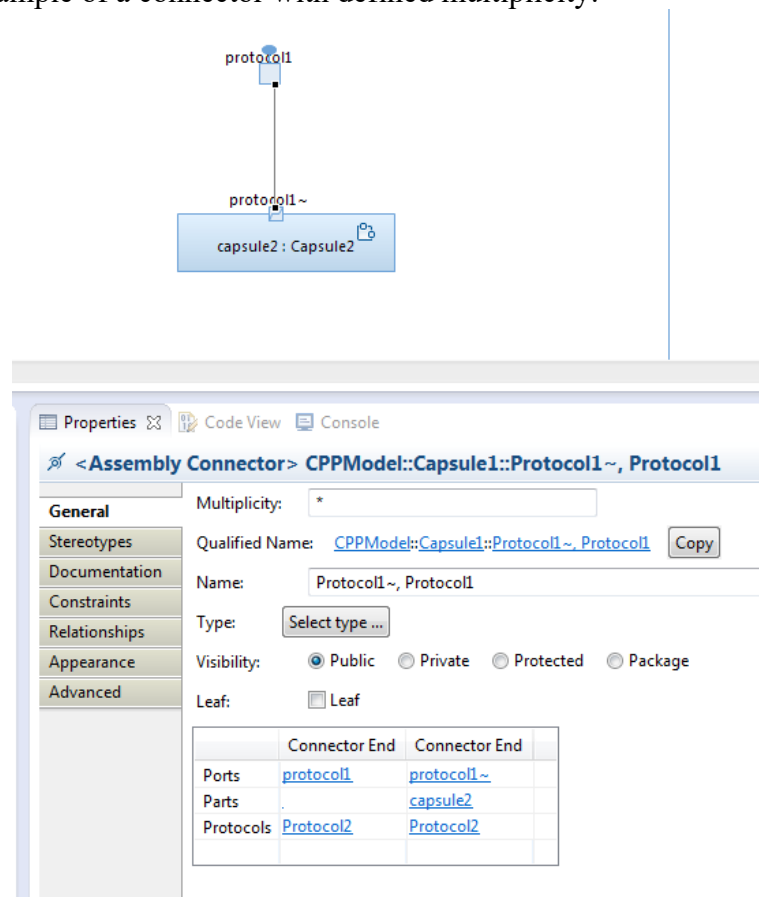
```

CPPModel::Collaboration1::Protocol1, Protocol1~.
Detected connector end with missing capsule part information in the connector
CPPModel::Collaboration1::Protocol1, Protocol1~2.
Detected connector end with missing capsule part information in the connector
CPPModel::Collaboration1::Protocol1, Protocol1~2.
Updating connector end of CPPModel::Collaboration1::Protocol1, Protocol1~.
Updating connector end of CPPModel::Collaboration1::Protocol1, Protocol1~2.
Updating connector end of CPPModel::Collaboration1::Protocol1, Protocol1~2.
Updating connector end of CPPModel::Collaboration1::Protocol1, Protocol1~.
4 connector end(s) has been updated.

```

Setting elements for multiplicity labels on structure diagrams

This fixup is related to incorrect multiplicity labels of connectors imported from Rose RT. It is recommended to run this fixup only if you have problems mentioned below when migrating diagrams from Rose RT. It should not be run periodically with already imported diagrams. Consider this example of a connector with defined multiplicity:



The diagram view (i.e. the line) that is associated with this semantic connector is represented in the model file as an edge with child nodes corresponding to connection view properties, including multiplicity label(s).

view definition

```

<edges xmi:type="umlnotation:UMLConnector" xmi:id="_Mnu629kUEeS3cPjeadz6Q"
source="_Mnu6ztkUEeS3cPjeadz6Q" target="_Mnu6idkUEeS3cPjeadz6Q" fontHeight="8"...>
.....

```

multiplicity labels definition

```

<children xmi:type="notation:DecorationNode" xmi:id="_Mnu64NkUEeS3cPjeadz6Q" visible="false" type="ToMultipli-
cityLabel">
<children xmi:type="notation:BasicDecorationNode" xmi:id="_Mnu64dkUEeS3cPjeadz6Q" type="ToMultiplicity"/>

```

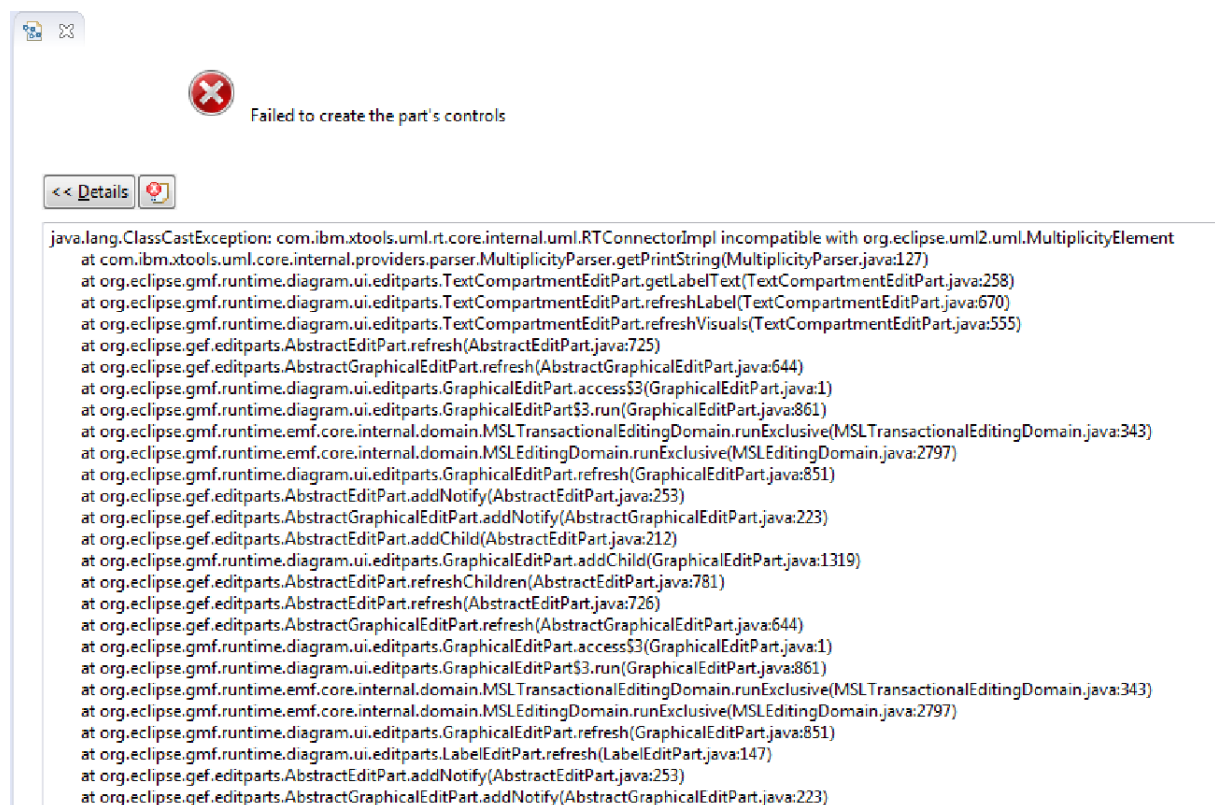
```

<element xmi:type="uml:ConnectorEnd" href="platform:/resource/RG_Model RealTimePackages/Function_Pack-
age.efx#_E5874J4DEeOm6Mbl35H6pQ?A_ConvenienceFunction/C4CBD816C01AD/%3CConnectorEnd%3E?"/>
<layoutConstraint xmi:type="notation:Location" xmi:id="_Mnu64tkUEeS3cPjeadz6Q" y="396"/>
</children>
<children xmi:type="notation:DecorationNode" xmi:id="_Mnu649kUEeS3cPjeadz6Q" visible="false" type="FromMultipli-
cityLabel">
<children xmi:type="notation:BasicDecorationNode" xmi:id="_Mnu65NkUEeS3cPjeadz6Q" type="FromMultiplicity"/>
<element xmi:type="uml:ConnectorEnd" href="platform:/resource/RG_Model RealTimePackages/Function_Pack-
age.efx#_E58U0Z4DEeOm6Mbl35H6pQ?A_ConvenienceFunction/C4CBD816C01AD/%3CConnectorEnd%3E?"/>
<layoutConstraint xmi:type="notation:Location" xmi:id="_Mnu65dkUEeS3cPjeadz6Q" y="396"/>
</children>

```

The nodes associated with the multiplicity labels should refer to the connector ends of the connector. This is required for correct view representation on composite structure diagram and all its inherited view representations as well.

It has been observed that after the import from Rose RT several connectors have missing semantic elements for its multiplicity labels. This may prevent the composite structure diagram from being possible to open:



This fixup determines if there are missing connector ends and corrects related model files. When you run this fixup make sure that your workspace contains all relevant projects (including those which contains related inherited elements).

Add missing package imports

Practically all applications make use of primitive C++ types (int, char, bool etc). To make these types available in your model (so they can be found when searching or using type completion), a package CppPrimitiveDatatypes is by default imported in all newly created Model RealTime models. However, models imported by the Rose RT importer may sometimes lack this package import. The consequence will be that references to primitive C++ types will be unbound, and that features such as type completion don't work for primitive types.

You can easily add a package import manually using the **Add UML - Relationship - Package Import** context menu command in the Project Explorer. However, if you have many models that lack this package import you can benefit from running this model fixup. In Analyze mode the fixup will report top-level packages it finds which lacks a package import to CppPrimitiveDatatypes. In Update mode the fixup will add the missing package import for those packages.

```
Applying fixup { Add missing package imports }.
Package Import - 'CppPrimitiveDatatypes' missing for resource
'TrafficLightComponent' with URI
'platform:/resource/TL/TrafficLightComponent.emx#_lBTt0KG6EeqtIqTcJcQviQ'
1 resources with missing imports found
Fixing in progress for missing import for 'TrafficLightComponent' with location
'platform:/resource/TL/TrafficLightComponent.emx#_lBTt0KG6EeqtIqTcJcQviQ' !!
Fixed 1 missing Package imports in { L/TL/TrafficLightComponent.emx }
Finished!
```

Use CDATA in model files

Starting with Model RealTime 11 2020.33 it's possible to save model files so that code snippets and documentation texts are stored in CDATA sections. This behavior is controlled by a preference *Modeling - Use CDATA when saving model files* which by default is not set. Using CDATA has the advantage that code and documentation from the model can be stored without any XML escape characters. This makes these texts more readable when viewing a model file in a text editor. Here is an example:

■ Example (without CDATA)

```
<headerPreface xmi:id="_irLY4NitEeqes4xYsqINTa"
body="// Some necessary
includes&#xD;&#xA;#include
<lt;iostream>&#xD;&#xA;#include
<lt;string>&#xD;&#xA;&#xD;&#xA;#define OPEN
1&#xD;&#xA;#define CLOSED 0"/>
```

■ Example (with CDATA)

```
<headerPreface xmi:id="_irLY4NitEeqes4xYsqINTa">
  <body><![CDATA[// Some necessary includes
#include <iostream>
#include <string>

#define OPEN 1
#define CLOSED 0]]></body>
</headerPreface>
```

This fixup helps with converting multiple model files to use CDATA, so that you don't need to save them one by one. It can be useful when you want to migrate everything to use the new file format, and to commit all changes at once.