Modelica Change Proposal MCP-0014 Conversion Status: Scheduled for Modelica 3.4

2015-10-15 version v1.1

1 Summary

This document documents how automatically convert library due to changes of used libraries.

2 Copyright License Revisions

2014-12-04 v1 - DRAFT	Hans Olsson: Initial (incomplete) draft version.
2015-12-09 v1.1 – DRAFT	Hans Olsson: Added specific conversion commands in the use cases
2015-10-15 v1.1 - BETA	Hans Olsson: Updated based on decisions at Modelica Design Meeting in Velizy

3 Proposed Changes in Specification

The precise specification changes could be further improved!

Please read the 18.8.2 has conversion annotations using a "script" with unspecified contents. The intent is to specify that and/or have another way of specifying conversions.

There are two new optional tags in the from-annotation, change and to:

```
conversion ( from (version = {VERSION-NUMBER,VERSION-NUMBER,…}, script = "…",
change={conversionCommand(), …, conversionCommand()}, to=VERSION-NUMBER)
```

Each from-annotation must specify either a script or a change (if no conversion is needed noneFromVersion can be used).

- The conversion script is comprised of call of the conversion commands specified in 18.8.2.1 where each call is terminated with semicolon.
- Each conversionCommand should be a call of one of the conversion commands defined in section 18.8.2.1.

The to-tag specifies which version to convert to (if the to-tag is missing it is assumed that conversion is to the current version); and is added for clarity and to allow optional multi-step conversions. The from-version-tags should be unique, indicating that based on an existing library version it is possible to determine which conversion to use.

18.8.2.1 Conversion commands

There are a number of conversion commands: convertClass, convertClassIf, convertElement, convertModifiers, convertMessage defined as follows.

```
convertClass("OldClass","NewClass")
// Convert class OldClass to NewClass
// Match longer path first, so if converting both A->C and A.B->D then A.B is
converted to D and A.B.E to D.E

convertClassIf("OldClass", "oldElement", "whenValue", "NewClass");
// Convert class OldClass to NewClass if the literal modifier for oldElement
// has the value whenValue, and also remove the modifier for oldElement.
//
// These are considered before convertClass and convertMessage
// for the same OldClass.

convertElement("OldClass","OldName","NewName")
// In OldClass convert element OldName (component or parameter) to NewName
// Both OldName and NewName are normally components (and in most cases
parameter/connectors) – but they may also be
// class-parameters, or hierarchical names.
// If conversion for both "inPort.signal" and "inPort" then convert longest match

convertModifiers("OldClass",{"OldModifier1=default1","OldModifier2=default2",...},{"Ne
wModifier1=...%OldModifier1%"}, simplify=false)
// Normal case; if any modifier among OldModifier exist then replace all of them with
the NewModifiers.
//
// The defaults (if present) are used if multiple OldModifiers and not all are set in
the component instance
// if simplify then perform obvious simplifications to clean up the new modifier
//
// Unusual cases:
// 1. if NewModifier list is empty then the modifier is just removed
// 2. If OldModifer list is empty it is added for all uses of the class
// Note: simplify is primarily intended for converting enumerations and emulated
enumerations
//       that naturally lead to large nested if-expressions.
//       The simplifications may also simplify parts of the original expression.
// 3. If OldModifier_i is cardinality(a)=0 the conversion will only be applied
// for a component comp if there is no connection to comp.a
// This can be combined with other modifiers that are handled in the usual way.
// 4. If OldModifier_i is cardinality(a)=1 the conversion will only be applied
// for a component comp if there is are any connections to comp.a

convertMessage("OldClass", "Failed Message");
// For any use of OldClass (or element of OldClass) report that conversion
// could not be applied with the given message.
// This may be useful if there is no possibility to convert a specific class.
// An alternative is to construct ObsoloteLibraryA for problematic cases.
```

These functions can be called with literal strings or array of strings and vectorize according to 12.4.6. The list here is the proposed subset of existing commands in Dymola (excluding some rare cases) – it is possible that we should add convertModifiers without the simplify-argument. At first it was considered to not add convertModifiers at all – but it seems to be genuinely useful, and thus should be added.

Both convertElement and convertModifiers only use inheritance among user models, and not in the library that is used for the conversion – thus conversions of base-classes will require multiple conversion-calls; this ensures that the conversion is independent of the new library structure. The class-name used as argument to convertElement and convertModifiers is similarly the old name of the class, i.e. the name before it is possibly converted by convertClass.

Motivation section to get an impression of the proposed changes. In the following a first specification draft is provided to stimulate discussion.

3.1 Required changes in List of Keywords

No change.

3.2 Required changes in Grammar

No change.

3.3 Required changes in Specification Text

18.8.2 has conversion annotations using a “script” with unspecified contents. The intent is to specify that and/or have another way of specifying conversions.

There are two new optional tags in the from-annotation, change and to:

```
conversion ( from (version = {VERSION-NUMBER,VERSION-NUMBER,…}, script = "…",
change={conversionCommand(), …, conversionCommand()}, to=VERSION-NUMBER)
```

Each from-annotation must specify either a script or a change (if no conversion is needed noneFromVersion can be used).

- The conversion script is comprised of call of the conversion commands specified in 18.8.2.1 where each call is terminated with semicolon.
- Each conversionCommand should be a call of one of the conversion commands defined in section 18.8.2.1.

The to-tag specifies which version to convert to (if the to-tag is missing it is assumed that conversion is to the current version); and is added for clarity and to allow optional multi-step conversions. The from-version-tags should be unique, indicating that based on an existing library version it is possible to determine which conversion to use.

18.8.2.1 Conversion commands

There are a number of conversion commands: convertClass, convertClassIf, convertElement, convertModifiers, convertMessage defined as follows.

```
convertClass("OldClass","NewClass")
// Convert class OldClass to NewClass
// Match longer path first, so if converting both A->C and A.B->D then A.B is
converted to D and A.B.E to D.E

convertClassIf("OldClass", "oldElement", "whenValue", "NewClass");
// Convert class OldClass to NewClass if the literal modifier for oldElement
// has the value whenValue, and also remove the modifier for oldElement.
//
// These are considered before convertClass and convertMessage
// for the same OldClass.

convertElement("OldClass","OldName","NewName")
// In OldClass convert element OldName (component or parameter) to NewName
// Both OldName and NewName are normally components (and in most cases
parameter/connectors) – but they may also be
// class-parameters, or hierarchical names.
// If conversion for both "inPort.signal" and "inPort" then convert longest match

convertModifiers("OldClass",{"OldModifier1=default1","OldModifier2=default2",...},{"Ne
wModifier1=...%OldModifier1%"}, simplify=false)
```

```
// Normal case; if any modifier among OldModifier exist then replace all of them with
the NewModifiers.
//
// The defaults (if present) are used if multiple OldModifiers and not all are set in
the component instance
// if simplify then perform obvious simplifications to clean up the new modifier
//
// Unusual cases:
// 1. if NewModifier list is empty then the modifier is just removed
// 2. If OldModifer list is empty it is added for all uses of the class
// Note: simplify is primarily intended for converting enumerations and emulated
enumerations
//       that naturally lead to large nested if-expressions.
//       The simplifications may also simplify parts of the original expression.
// 3. If OldModifier_i is cardinality(a)=0 the conversion will only be applied
// for a component comp if there is no connection to comp.a
// This can be combined with other modifiers that are handled in the usual way.
// 4. If OldModifier_i is cardinality(a)=1 the conversion will only be applied
// for a component comp if there is are any connections to comp.a

convertMessage("OldClass", "Failed Message");
// For any use of OldClass (or element of OldClass) report that conversion
// could not be applied with the given message.
// This may be useful if there is no possibility to convert a specific class.
// An alternative is to construct ObsoloteLibraryA for problematic cases.
```

These functions can be called with literal strings or array of strings and vectorize according to 12.4.6. The list here is the proposed subset of existing commands in Dymola (excluding some rare cases) – it is possible that we should add convertModifiers without the simplify-argument. At first it was considered to not add convertModifiers at all – but it seems to be genuinely useful, and thus should be added.

Both convertElement and convertModifiers only use inheritance among user models, and not in the library that is used for the conversion – thus conversions of base-classes will require multiple conversion-calls; this ensures that the conversion is independent of the new library structure. The class-name used as argument to convertElement and convertModifiers is similarly the old name of the class, i.e. the name before it is possibly converted by convertClass.

4 Motivation

In order to allow libraries to be improved classes and components have to be renamed and/or changed in other incompatible ways. User models should (whenever possible) be automatically be updated to reflect these changes; in order to perform this in general the library should declare what changes are needed in user models.

4.1 Use Cases

The trac tickets with milestone “MSL_withConversionScript” contain several future examples; and there are many existing use cases in:
modelica://Modelica/Resources/Scripts/Dymola/ConvertModelica_from_2.2.2_to_3.0.mos

The use case are all variations of the theme of improving the library, and automatically updating models to stay compatible.

Correct bad spelling

In current MSL there are some parameters named “mue” and “lamda” instead of the correct names. The proposed conversion allows the spelling to be corrected (to “mu” and “lambda”) and automatically updating user models to ensure that they stay compatible; and similar for other spelling corrections.

```
convertElement({"Modelica.Mechanics.Rotational.Components.Brake",
"Modelica.Mechanics.Rotational.Components.Clutch",
   "Modelica.Mechanics.Rotational.Components.OneWayClutch"}, "mue_pos", "mu_pos");

convertElement({"Modelica.Thermal.FluidHeatFlow.Media.Medium",
"Modelica.Thermal.FluidHeatFlow.Media.Air_30degC",
   "Modelica.Thermal.FluidHeatFlow.Media.Air_70degC",
"Modelica.Thermal.FluidHeatFlow.Media.Water"}, "lamda", "lambda");
```

Restructure library

Sometimes it is discovered that there are too many models in a package or that some of them should not be there and it would be better to move them to different sub-packages. An example in version 2 of MSL was Modelica.SIunits.CelsiusTemperature; which is not an SIunit and thus was moved to better location in version 3 of MSL; and similarly Modelica.Mechanics.Rotational did in version 2 of MSL have components directly in the library – which at the start was ok, but when more models accumulated it was not problematic and was therefore improved in version 3 by moving those models to one of Components, Sources, and Sensors. The user models can for such cases be updated automatically by the proposed conversions.

```
convertClass("Modelica.SIunits.CelsiusTemperature",
             "Modelica.SIunits.Conversions.NonSIunits.Temperature_degC");

convertClass  ("Modelica.Mechanics.Rotational.Position",
"Modelica.Mechanics.Rotational.Sources.Position");
convertClass  ("Modelica.Mechanics.Rotational.Speed",
"Modelica.Mechanics.Rotational.Sources.Speed");
convertClass  ("Modelica.Mechanics.Rotational.Accelerate",
"Modelica.Mechanics.Rotational.Sources.Accelerate");
convertClass  ("Modelica.Mechanics.Rotational.Move",
"Modelica.Mechanics.Rotational.Sources.Move");
convertClass  ("Modelica.Mechanics.Rotational.Torque",
"Modelica.Mechanics.Rotational.Sources.Torque");
convertClass  ("Modelica.Mechanics.Rotational.Torque2",
"Modelica.Mechanics.Rotational.Sources.Torque2");
convertClass  ("Modelica.Mechanics.Rotational.LinearSpeedDependentTorque",
   "Modelica.Mechanics.Rotational.Sources.LinearSpeedDependentTorque");
convertClass  ("Modelica.Mechanics.Rotational.QuadraticSpeedDependentTorque",
   "Modelica.Mechanics.Rotational.Sources.QuadraticSpeedDependentTorque");
convertClass  ("Modelica.Mechanics.Rotational.ConstantTorque",
"Modelica.Mechanics.Rotational.Sources.ConstantTorque");
convertClass  ("Modelica.Mechanics.Rotational.TorqueStep",
"Modelica.Mechanics.Rotational.Sources.TorqueStep");
convertClass  ("Modelica.Mechanics.Rotational.ConstantSpeed",
"Modelica.Mechanics.Rotational.Sources.ConstantSpeed");
convertClass  ("Modelica.Mechanics.Rotational.RelativeStates",
   "Modelica.Mechanics.Rotational.Components.RelativeStates");
convertClass  ("Modelica.Mechanics.Rotational.Fixed",
"Modelica.Mechanics.Rotational.Components.Fixed");
convertClass  ("Modelica.Mechanics.Rotational.Inertia",
"Modelica.Mechanics.Rotational.Components.Inertia");
```

```
convertClass  ("Modelica.Mechanics.Rotational.Spring",
"Modelica.Mechanics.Rotational.Components.Spring");
convertClass  ("Modelica.Mechanics.Rotational.Damper",
"Modelica.Mechanics.Rotational.Components.Damper");
convertClass  ("Modelica.Mechanics.Rotational.SpringDamper",
"Modelica.Mechanics.Rotational.Components.SpringDamper");
convertClass  ("Modelica.Mechanics.Rotational.ElastoBacklash",
   "Modelica.Mechanics.Rotational.Components.ElastoBacklash");
convertClass  ("Modelica.Mechanics.Rotational.BearingFriction",
   "Modelica.Mechanics.Rotational.Components.BearingFriction");
convertClass  ("Modelica.Mechanics.Rotational.Brake",
"Modelica.Mechanics.Rotational.Components.Brake");
convertClass  ("Modelica.Mechanics.Rotational.Clutch",
"Modelica.Mechanics.Rotational.Components.Clutch");
convertClass  ("Modelica.Mechanics.Rotational.OneWayClutch",
"Modelica.Mechanics.Rotational.Components.OneWayClutch");
convertClass  ("Modelica.Mechanics.Rotational.IdealGear",
"Modelica.Mechanics.Rotational.Components.IdealGear");
convertClass  ("Modelica.Mechanics.Rotational.LossyGear",
"Modelica.Mechanics.Rotational.Components.LossyGear");
convertClass  ("Modelica.Mechanics.Rotational.IdealPlanetary",
   "Modelica.Mechanics.Rotational.Components.IdealPlanetary");
```

Improve parameterization

In some cases there is an issue with the chosen parameters, e.g. in version 2 of MSL the density for Modelica.Mechanics.MultiBody.Parts.BodyBox was specified using g/cm^3 instead of kg/m^3; the user models can for such cases be updated automatically by the proposed conversions – using convertModifiers.

```
convertModifiers("Modelica.Mechanics.MultiBody.Parts.BodyBox"     ,{"density"},
{"density=%density%*1000"});
```

Remove parameters

Sometimes a parameter is removed from a model without impacting the use of the model, e.g. visualization parameters or the SignalType used for block-connectors in version 2 of MSL; and sometimes parameter a start-value “phi_start” is removed and the user required to directly modify “phi.start” instead. The user models can for such cases be updated automatically by the proposed conversions – using convertModifiers relying on the specified handling of empty list for the new modifier.

```
convertModifiers("Modelica.Mechanics.Rotational.Inertia",{"phi_start"},
{"phi.start=%phi_start%"});

convertModifiers("Modelica.Blocks.Interfaces.RealInput",{"SignalType"},fill("",0),true
);
convertModifiers("Modelica.Blocks.Interfaces.RealOutput",{"SignalType"},fill("",0),tru
e);
convertModifiers("Modelica.Blocks.Interfaces.RealSignal",{"SignalType"},fill("",0),tru
e);
```

Merging models

In version 2 of MSL the thermal library had two variants of each source – one with an input and one with a parameter. Having two variants makes it difficult for the user to find the appropriate one – and in version 3 this was instead controlled using a parameter. The user models can for such cases be updated automatically by the proposed conversions – using convertModifiers with the specified handling of empty list for the old modifiers.

```
convertClass    ("Modelica.Thermal.FluidHeatFlow.Sources.Ambient",
"Modelica.Thermal.FluidHeatFlow.Sources.Ambient");
convertModifiers({"Modelica.Thermal.FluidHeatFlow.Sources.Ambient"}, fill("",0),
  {"usePressureInput=false", "useTemperatureInput=false"});
convertElement  ({"Modelica.Thermal.FluidHeatFlow.Sources.Ambient"}, "p_Ambient",
"constantAmbientPressure");
convertElement  ({"Modelica.Thermal.FluidHeatFlow.Sources.Ambient"}, "T_Ambient",
"constantAmbientTemperature");

convertClass    ("Modelica.Thermal.FluidHeatFlow.Sources.PrescribedAmbient",
"Modelica.Thermal.FluidHeatFlow.Sources.Ambient");
convertModifiers({"Modelica.Thermal.FluidHeatFlow.Sources.PrescribedAmbient"},
fill("",0),
   {"usePressureInput=true", "useTemperatureInput=true"});
convertElement  ({"Modelica.Thermal.FluidHeatFlow.Sources.PrescribedAmbient"},
"p_Ambient", "ambientPressure");
convertElement  ({"Modelica.Thermal.FluidHeatFlow.Sources.PrescribedAmbient"},
"T_Ambient", "ambientTemperature");
```

Splitting models

Sometimes the different values for a (Boolean) parameter are sufficiently different that they should be seen as different models – or one (rare) case will no longer be supported.

```
convertClass  ("Modelica.Mechanics.MultiBody.Joints.ActuatedRevolute",
               "Modelica.Mechanics.MultiBody.Joints.Revolute");

convertClassIf("Modelica.Mechanics.MultiBody.Joints.ActuatedRevolute",
"planarCutJoint","true",
               "Modelica.Mechanics.MultiBody.Joints.RevolutePlanarLoopConstraint");

convertClassIf("Modelica.Mechanics.MultiBody.Joints.Internal.PrismaticWithLengthConstr
aint","axisForceBalance","false",
"Modelica.Mechanics.MultiBody.Joints.Internal.PrismaticWithLengthConstraint");

convertClassIf("Modelica.Mechanics.MultiBody.Joints.Internal.PrismaticWithLengthConstr
aint","axisForceBalance","true",
"ObsoleteModelica3.Mechanics.MultiBody.Joints.Internal.PrismaticWithLengthConstraint"
);

convertClass
("Modelica.Mechanics.MultiBody.Joints.Internal.PrismaticWithLengthConstraint",
"ObsoleteModelica3.Mechanics.MultiBody.Joints.Internal.PrismaticWithLengthConstraint"
);
```

In this case existing components of ActuatedRevolute with the modifier planarCutJoint=true will be converted to RevolutePlanarLoopConstraint (and the planarCutJoint-modifier removed). Without the modifier they will be converted to Revolute.

Existing components of PrismaticWithLengthConstraint with axisForceBalance=false will be converted to the PrismaticWithLengthConstraint (same name) and axisForceBalance-modifier removed; existing components of PrismaticWithLengthConstraint with axisForceBalance=true, or without axisForceBalance-modifier will be converted to use an obsolete package (as a temporary solution).

Alternative to Obsolete-package

Above complicated cases were converted to an Obsolete-package. The intent is to keep models running after conversion, and encourage users to later convert from the Obsolete-package to better models. The models in the Obsolete-package would normally use the obsolete-annotation to indicate why they are no longer suited, and what the user should do.

In some cases that may not be necessary and a simpler alternative is then to use convertMessage:

```
convertClassIf("Modelica.Mechanics.MultiBody.Joints.Internal.PrismaticWithLengthConstr
aint","axisForceBalance","false",
"Modelica.Mechanics.MultiBody.Joints.Internal.PrismaticWithLengthConstraint");

convertMessage("Modelica.Mechanics.MultiBody.Joints.Internal.PrismaticWithLengthConstr
aint","Cannot handle axisForceBalance unless false – see documentation.");
```

In this case problematic models will just generate a diagnostics during conversion.

Removing cardinality from models

Models using cardinality internally are problematic, both because reflection is problematic – and also because cardinality is behaving in unexpected ways often leading to incorrect behavior.

```
convertModifiers({"Modelica.Mechanics.MultiBody.Sensors.CutForce"},
                 {"cardinality(frame_resolve)=0", "resolveInFrame_a=true"},
                 {"resolveInFrame = if (%resolveInFrame_a%) then " +
     "Modelica.Mechanics.MultiBody.Types.ResolveInFrameA.frame_a else " +
     "Modelica.Mechanics.MultiBody.Types.ResolveInFrameA.world"}, true);

convertModifiers({"Modelica.Mechanics.MultiBody.Sensors.CutForce"},
{"cardinality(frame_resolve)=1"},
{"resolveInFrame=Modelica.Mechanics.MultiBody.Types.ResolveInFrameA.frame_resolve"});
```

In this case the result is that for an existing component of CutForce the resolveInFrame is set as follows:

- ResolveInFrameA.frame_resolve, if frame_resolve is connected.
- ResolveInFrameA.frame_a, if frame_resolve is un-connected and resolveInFrame_a=true (default)
- ResolveInFrameA.world, if frame_resolve is un-connected and resolveInFrame_a=false (non-default).

The design of this special case for cardinality is far from ideal. However, it was the existing solution for converting from Modelica 2 to 3 (implemented in Dymola), and hopefully conversions involving cardinality will be a temporary solution to correct some models in Modelica 3 and then phased out.

4.2 Impact

The immediate impact of having conversions specified in the language is to document the existing practice of upgrading user models in Dymola. The next impact is that libraries (such as Modelica Standard Library) can be improved more freely since the library developer can rely on user models being updated.

5 Rationale

5.1 Design process

The design of some conversion was needed in Dymola to convert from the Dymola language to Modelica; it was also used to convert user models from Modelica 1 to Modelica 2 and then from Modelica 2 to 3. Each of these conversions also required some special handling that is not part of the proposal – we might similarly have to consider some special primitives to handle the next set of conversions.

5.2 Alternative Designs

The design uses a declarative list of what conversions to perform, and requires the tool to automatically perform them.

A variant of the proposal would be that the conversions are ordered, e.g. if use convertClass(“A”, …) and convertClass(“A.B”, ….) the second one would not automatically be used for A.B, but would the longer path to be placed earlier in the list; and if there was a convertElement(“A.B”, …) it would only be used if it was placed before the convertClass calls. Such a list would be easier to build automatically, and would easily allow multi-stage conversions. On the other hand it will take more care to maintain the conversions manually (it seems that diagnostics for “shadowed” conversions would be needed), and require that the order is specified for vectorized calls. It has not been analyzed if these variants differ in terms of what is possible to do.

The proposed declarative list could be given in other formats, e.g. as XML-file; but it does not seem to give any advantage since the rewrite cannot be done with a simple XML-rewrite – and the tool performing the conversion must anyway be able to parse and semantically handle Modelica. The commands in Modelica can also follow the normal Modelica rules for quoting of strings and naturally allow vectorization.

A completely different possibility would be to require a user models to perform the rewrite by some yet unspecified syntax-tree API; but that would require substantially more work (in order to specify a syntax-tree API that allows both lookup and modifying the code; and also write routines to perform the conversions). The procedural handling of the conversion would limit the conversions to the considered use-cases, e.g. test-running the conversion would not be easy with such a design. On the other hand if/when such an API-interface and a reusable library for conversions were implemented it could be used to implement the proposed design.

Another different possibility for these issues is to maintain the old interface forever – and have some way of supporting a larger interface for Modelica classes (aliases for classes and connectors/parameters); that would be similar to performing the conversion every time the old interface is used which would be difficult to maintain for each library, and develop support for in tools (to hide the old interface for new use). This design also normally pre-supposes a separation of implementation and interface that is not possible in Modelica (such that the same implementation could easily be reused to implement different interfaces), but is common for other APIs.

5.3 State of the Art

The proposal is already implemented in Dymola several years ago; and the proposal is the result of several iterations.

6 Backwards Compatibility

The proposed solution is backwards compatible.

Furthermore it allows user models to retain backwards compatible with libraries for more changes than previously possible.

6.1 Corruption of existing models

The proposal does not in itself corrupt any existing models.

However, a bad library could define incorrect conversions that would corrupt existing models, similarly as the new library version could contain incorrect models.

6.2 Means of mitigation

The conversion can prompt the user before upgrading and automatically create a backup of the old user models to mitigate the problem of incorrect conversions. Normal practice is also to verify the models after conversions, and keep Modelica packages in a version control system such that an incorrectly converted model can be restored.

The conversion commands are a declarative list of conversions that could be inspected.

7 Implementation Effort

Fairly small for a tool that already implements the Modelica semantics.

7.1 Mandatory Parts

The conversion commands and vectorization of them.

7.2 Optional Parts

Automatically generating the conversion commands while editing (only possible for convertClass and convertElement); an alternative is that the user manually writes the conversion commands needed in their library. A non-prototyped optional part would be to verify the completeness of the conversions against the different versions of the library.

Checking if any conversion is necessary for a specific model by test-running the conversion; an alternative is to always convert.

Multi-stage conversion, i.e. automatically running conversions from version 1 to 2 and then to 3 without having access to version 2. This relies on the optional to-tag defining what the conversion converts to, and there is another conversion matching the to-version. This must be done in a way that correctly handles the case where the model is renamed in both conversions (or modified in other ways); an alternative is to require the user to first upgrade from 1 to 2 (using version 2) and then from 2 to 3.

7.3 Way of Implementation

Prototyped in Dymola; traverse all classes in the user library using the new library version and use a variant of normal lookup to see if models reference modified elements (without the older version of the library the normal lookup will fail for any renamed elements).

Another implementation (not prototyped) would require loading the older version of the library and could then perform normal lookup for all user classes. This seems simpler to implement, but more difficult to use since the conversions also require the old version.

8 References

[1] Dymola User’s guide.