Modelica 3.6 - Changes, Benefits and Implementation

Hans Olsson[1]

[1] Dassault Systèmes, Sweden, hans.olsson@3ds.com

Abstract

The latest release of the Modelica Language Specification version 3.6 brings several benefits to users, and this paper will discuss the changes and the benefits for the clearer parameter defaults, clearer start-value priority, selective model extension, and multi-lingual support. The benefits only occur when the features are implemented in Modelica tools, and to facilitate that, the paper will discuss the design choices when implementing the new standard in Dymola 2023x Refresh 1 and 3DEXPERIENCE 2023x FD03.

Keywords: Modelica, model variation, initialization

1 Introduction

The Modelica Language, (Olsson (editor), 2023) and (MAP-Lang 2023), is developed by the Modelica Association Project MAP-Lang on GitHub, using LaTeX and HTML (Miller 2023).

Modelica 3.6 adds a number of new features, corrections, and improvements. New major features are organized as Modelica Change Proposals (MCPs), specifically for this release:

- Undefined modification (MCP-0009).
- Selective Model Extension (MCP-0032).
- Multilingual support of Modelica (MCP-0035).

Their rationales can be found in the directory RationaleMCP in (MAP-Lang 2023). In the specification they can be found in sections 7.2.7, 7.4, and 13.6.

An MCP must be test-implemented in at least one tool before being added to the specification, and the design documents include that experience.

However, fully implementing, documenting, and testing an MCP may reveal new issues (especially when done in other tools than the tools used for the test implementations) and may also find interesting use-cases; and thus the experience in this paper will add additional insights beyond the MCP-design.

This paper will focus on the new major features (MCPs), and some of the minor features and corrections; especially concerning parameters (that go together with the MCP-0009 Undefined modification), start-value priority, and connection restrictions.

Modelica 3.6 was completed on February 28th, the document branch built March 9th, and accepted by Modelica Association March 23rd, 2023.

2 Clear setting of parameters

2.1 Background

MCP-0009 (Undefined modification) goes together with some minor improvements (and corrections) related to parameter values and defaults. The MCP was proposed by ESI (previously ITI) whereas the corrections and minor improvements roughly correspond to what was already implemented in Dymola.

Prior to Modelica 3.6 the specification had the following issues:

- Once you had set a parameter value you could only change it, but not remove the setting completely. Removing setting is useful when:
  - A model parameter has a badly chosen default value and there is no obvious generic correct value (e.g., capacitance).
  - A model parameter has a value, but it is desired to implicitly compute the parameter from an initial equation instead. (Note: in this case it is also necessary to set `fixed=false`.)
- If you declared a parameter without modifying any of its attributes (i.e., no start-value) it was unclear what tools should do; whereas if you did set the start-value it was clear that it could be used with a warning.
- And if you declared a parameter (or variable) and only set the min-attribute to e.g., `2` it was even less clear what tools should do.

The reason for the latter two problems were that the default value for the start-attribute of a Real/Integer was `0`, and it was not clear whether that should be used as default.

A model with all of these issues is given below:

```
model M
  parameter Integer k=1;
  parameter Integer l(start=1)
  parameter Integer m;
  parameter Integer n(min=2);

  Real x[:]=fill(0.0, n-1);
end M;

model M2
  extends M;
…
end M2;
```

Prior to Modelica 3.6 we could not remove the default for `k` when constructing `M2` and it was unclear if `m` should use a default start-value of `0` or not, and clearly using a default start-value of `0` for `n` would be problematic: by violating the min-restriction and generating a negative sized array.

2.2 Changes

MCP-0009 (Undefined modification) solves the first issue by allowing the modifier `=break` to remove the parameter setting. The other issues were solved by clearly specifying a priority for default values and erasing the default values for the start-attributes in the specification and instead introducing the fallback value.

The fallback value is the value closest to "zero" that is consistent with any potential min and max-attribute (for a Boolean it is `false` and for a String it is `""`); adding this restriction ensures internal consistency so that if a variable has a min-attribute of `2` we do not attempt to use a value of `0` (which solves the last issue).

Thus Modelica 3.6 allows us to give a clear precedence for different values for a parameter (unless they have fixed=false) as follows:

1. Value (unless Undefined).
2. Value of the start-attribute (unless Undefined).
3. Fallback value during check.

The default for both the Value and the Value of the start-attribute is now Undefined (which means that the next item in the precedence list is used), and MCP-0009 enables a user to restore them to Undefined by a modification of the form `=break`. A diagnostic is required when the Value is Undefined in a simulation model.

Note that `break` is just for modifiers, and it is not possible to use it to handle other variants of Undefined. Thus it does not correspond to Not-A-Number in IEEE floating point arithmetic, the Maybe-monad in functional languages, or std::optional in C++ (ISO/IEC 2020).

For a non-parameter that is used in an initial non-linear system of equations the starting point for the first iteration is:

1. Value of the start-attribute.
2. Fallback value.

2.3 Implementation aspects

Undefined modification can be fairly trivially implemented in the translator by treating `break` as a normal modification with the special rule that if the resulting modification after merging is `break` the setting of the value is just skipped.

Supporting it in the Graphical User Interface was not complicated, but required care to keep existing options as before *and* clearly specify the new option. If there is a modifier for a parameter `p` like `p=2` the parameter dialog shall support both removing the modifier giving no modification at this level or setting `p=break`. Both could be described in similar ways, which would not be helpful.

The solution was to call the first "Remove modifier" (as earlier) and the second "Set to No Value"; where "No Value" was preferred over the specification word "Undefined" and the syntactic "break". Obviously the user can also write these as modifiers in the parameter dialog.

For backwards compatibility there is an issue, not with Modelica 3.5 but going back to Modelica 2.0 where a component could be named `break` and thus `p=break` could mean that `p` is modified to have the value of the parameter `break`. That was solved when supporting Modelica 2.1 by handling existing components named `break` with a warning (that was possible in Modelica 2.1 to 3.5 since `break` was then only used to break inside loops), and prevent the creation of new components named `break`. That compatibility feature was removed after about 11 years (consistent with the mission plan of MAP-Lang where models should run for at least 10 years), and before Modelica 3.6 was released.

Using the fallback value when checking a model is consistent with the intended use – zero, or close to zero, and consistent with min- and max-attributes. In practice it is almost always the min-attribute that introduces the restriction, since if a variable cannot have both signs the preference is to have only non-negative values (e.g., temperatures, array sizes) and even if further restricted the min-attribute is the allowed value closest to zero.

3 Clearer precedence for start-values

3.1 Background

Modelica is equation based and in order to efficiently handle the resulting systems of equations tools have to make a number of choices for start-values in systems of equations and during initialization – this indirectly also influences which variables are torn out. For a general introduction to tearing see (Elmqvist & Otter 1994).

Consider Modelica.Fluid.Examples.HeatingSystem:

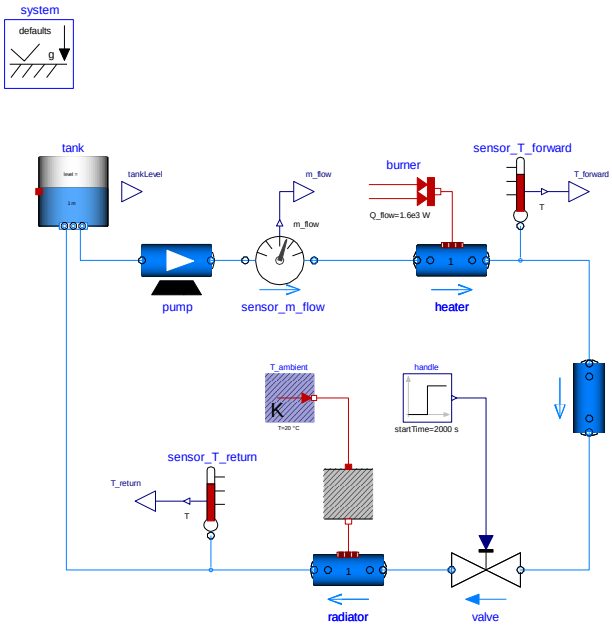

Figure 1 HeatingSystem

DOI
10.3384/ecp204367

There are 4 pressure states and 5 temperature states in this model, and for each of them there are between 7 and 13 different variables that are equal to the state, but have their own start-value. Setting all of them to the same value is tedious and error-prone, but partially done in this model:

```
Pipes.DynamicPipe radiator(
    …
    p_a_start=110000,
    state_a(p(start=110000)),
    state_b(p(start=110000)))
```

These choices matter as different choices can lead to different results or even no results, when reusing a Modelica model in a different tool, different version of the same tool, or even as a sub-model in a different model.

Using the average of the values does not work (e.g., for this model the default temperature in the components was 15°C and the goal was to change the hot part to 80°C – not to have an average of them), and having an explicit priority system was deemed too complicated so instead the idea is to prioritize the existing start-values in some logical way.

The goal with that priority is both to make the choice more predictable (so that the same model gives the same choice) and controllable for users (to get the start-values they want).

3.2 Start-value priority

Modelica 3.3 introduced a priority between start-values with the clear goal that values set "later" (closer to the root of the instance hierarchy) should have precedence. This was based on existing heuristics in Dymola (Dassault Systemes 2023 section 5.8.3; Casella 2011). The rationale with preferring a "later" value is that if there is a problem the user should introduce a new start-value when using the component and that will naturally be seen as "later".

However, it was found wanting in some cases – in particular start-values are often bound to parameters that are then propagated, e.g., in the model above the start-value for the temperature of the heater is the parameter `T_start`. The priority for such cases was based on where the parameter was introduced (and propagated to the start-value) – ignoring whether the parameter was modified later on. In practice that often meant that multiple values had the same priority.

Modelica 3.6 extended the priority to consider where the parameter is set (in this case `T_start`) to break ties in such cases. By only using it as a tie-breaker it adds more detailed priorities without modifying the priority of existing unambiguous cases, reducing the risk of breaking backwards compatibility.

3.3 Implementation aspects

Implementing a more detailed priority for guess-values was fairly straightforward (there were some existing special cases that had to be removed as well). And even if designed with backwards compatibility in mind it is also possible to disable the new feature in Dymola. (Having a standardized way of disabling the new feature was not considered. It would add unnecessary complexity as it is always possible to resolve issues in specific models by adding new start-values with higher priority.)

However, even if the priority is predictable and controllable for users an additional requirement is that the choice is explainable. Thus the logging of start value priority was improved to provide those priorities and all considered start-values. In this model enabling logging gives:

The iteration variable heater.mediums[1].T has been selected to have the guess value 353.15.

- 353.15, the start value of heater.mediums[1].T given as heater.T_start. At level 1. Original start-value at level 2.
- 288.15, the start value of heater.flowModel.states[2].T given as 288.15.
- …

The place where T_start was modified gives the level, whereas the original start-value level is where heater.mediums[1].T.start was modified. Levels are counted up from the current model and thus a lower level has precedence.

4 Selective Model Extension

MCP-0032 introduces selective model extension as a way of selectively deciding what to inherit from another model, and uses the same keyword `break` as MCP-0009 with similar considerations.

4.1 Goal of Selective Model Extension

The goal of selective model extension was originally to enable unforeseen structural variation by giving the possibility to exclude components and connections when inheriting, and doing it in a traceable and well-defined way, (Bürger 2019).

It has also been found useful when the structural variations *could be foreseen*, but supporting all possible foreseen structural variations would create a too complicated model.

Automatically generated models e.g., for Mechanical systems (Elmqvist et al 2009) and Fluid, is thus an additional use-case where a model can be reproducibly generated (so that the physical models always have the correct parameters), and then some connections selectively deselected and control components added. As an example if the previously shown HeatingSystem had been automatically generated from a physical model all of the sensors and actuator had likely been missing. Adding the flow-sensor would be a typical case where it is necessary to deselect a connection to insert a new component (and two new connections).

4.2 Implementation aspects

4.2.1 Deselecting connections

The original test-implementation of MCP-0032 handled deselections in the translator and when showing classes in the Graphical User Interface, but the deselections were written textually (despite the idea naturally being described in terms of changes of the diagram).

What was missing was the User Experience of actually deselecting graphically. This was done by adding "Deselect" option to the context menu of inherited elements (currently with a warning), similarly as the "Delete" option.

That revealed an unforeseen case – a user might first deselect a connection and then later deselect one of its endpoint components (or attempt to deselect both the connection and the component at the same time). That is an error according to the specification (since it does not make sense to write that textually), and can be avoided by adding an extra step removing redundant deselections of connections after any change of the deselections.

4.2.2 Automatic connector sizing

Applying deselections to Fluid models revealed that it interacted with connector sizing in unforeseen ways.

Fluid models have arrays of connectors with automatic sizing (introduced in Modelica 3.1) which ensures that different connections to the array are treated as multiple independent connectors, and the array is adapted in size. Treating them as independent connectors allows correct mixing for stream-connectors (Franke et al 2009); and is normally not necessary in other domains.

Note it is possible to use automatic sizing for other domains – one well-known use is multi-input logical And/Or-blocks; another use is to add parameter-attributes to each array element for a physical array of connectors; similar considerations apply in those cases.

Before Modelica 3.6 automatic sizing always created a dense array of connectors, and if you removed the connection to an element in the middle of the array the array was shrunk and elements re-numbered. Note that the component with automatic sizing connector can be inherited and local connections added – but the inherited connections always precede them.

However, when deselecting it is possible to remove an inherited connection to an element in the middle of an array of connections – and it is not straightforward to re-number the remaining inherited connections. More importantly it is usually not *desirable* to re-number them, as the intended use of Selective Model Extension is usually to re-introduce another connection to the same connector element (after adding/removing some component in the path of the fluid).

As an example look at the reusable HeatingSystem model, and consider deselecting the connection between radiator and tank to add an additional radiator.

The new model uses deselections:

```
model HeatingSystem2
  extends HeatingSystem(break
    connect(radiator.port_b,
            tank.ports[1]));
  Modelica.Fluid.Pipes.DynamicPipe
                    radiator1(…);
  …
equation
  connect(radiator1.port_b, tank.ports[1]);
  …
end HeatingSystem2;
```

In the diagram we can see an additional radiator (with sensor and wall-components), and that the new connection to the tank replaces the existing one.

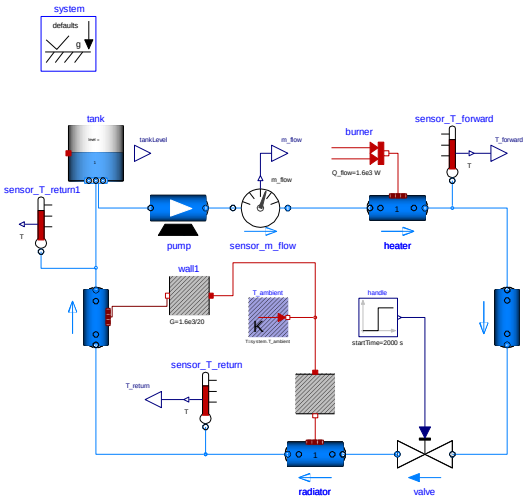

Figure 2 HeatingSystem with extra radiator.

Instead of renumbering the connections this is accomplished by modifying the User Experience of graphically connecting to a connector with automatic sizing to check if there are any holes due to deselected connections (or components) and suggest connecting to the missing element(s) instead of automatically adding it to the end and resizing the connector.

4.2.3 Possible extensions

The current selective model extension works for graphical objects: connections and components.

Deselecting non-connect equations is not possible as equations are not named (a potential extension) and deselecting a component used in such equations does not work either. This is not entirely trivial as one problem with even deselecting connect-equations is that the deselection are *by design* sensitive to the exact syntax used in the equations; and for non-connect equations this problem gets worse. However, when models are structured with large textual equations it may be useful. Automatically de-selecting equations would also be useful for removing e.g.,

DOI
10.3384/ecp204367

initial equations for de-selected components – but it may be possible to find another solution for that.

Additionally even for graphical objects there are some possible extensions. When filtering a signal it is currently necessary to de-select the connection, add the filter, and then reconnect it on both sides. Similarly when using selective model extension to replace a non-replace component it is necessary to first deselect the component and then add a new component and connect it. Simplifying that would be possible (a tool might possibly add this without modifying the language). On the other hand making such operations too easy might risk errors – and could lead to under-use of replaceable component, and relying on replacing them in this way.

4.2.4 Implementation variants

The flattening in Modelica is a hierarchical tree traversal where modifications are propagated downwards, and the resulting tree is then transformed into a hybrid DAE that is simulated.

The deselection is in the MCP seen as deselecting (or pruning) sub-trees after they have been built. That ensures that the deselection actually deselects something and that the pre-deselection elements are correct; and can also be implemented in a straightforward way in the translator.

Propagating deselections downwards similarly as modifications and preventing them from being built does not easily allow similar checks, but was implemented for the Graphical User Interface.

The benefit is that it allows the components to efficiently directly draw their graphics, instead of generating an intermediate graphical representation that is later pruned, and it also allows treating deselected components uniformly with normal components. A uniform treatment of all components in the component browser allows a toggle for deselecting components – to both show the current status and revert deselections. Something similar may be implemented for connections in the future.

5 Connection Restrictions

Causal connectors (with input and/or output) have restrictions to ensure that any input must be given a value - they normally imply that if there is an input component in the connector it must be connected exactly once from the outside (Olsson et al 2008); before Modelica 3.0 this rule only applied to entire connectors declared as input.

Modelica 3.3 added the restriction that conditional physical connectors (i.e., connectors with at least one flow variable) must be connected if enabled. The idea was that if you set a Boolean parameter to enable that connector it would not make sense to leave it unconnected, and the default semantics for unconnected connectors (zero flow) do not always make sense.

The intended case was the optional support connectors in the rotational library where many components have an optional support connector. The default (top part of diagram) is that it is disabled and instead there is an implicit connection to ground (giving zero position instead of zero flow) – but if enabled (bottom part) there is a new connector for the support of the component. The connector is marked with a red circle and the connections are red and dashed.

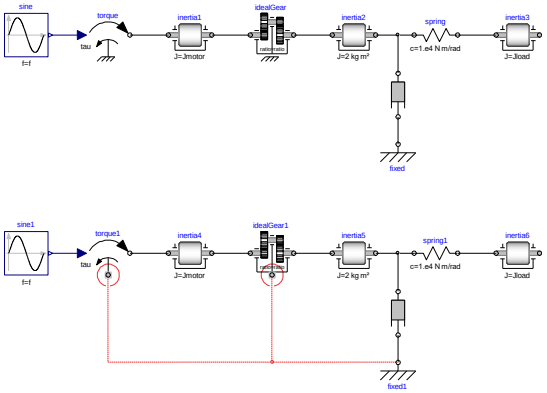

Figure 3 Driveline with implicit grounding (top) and with explicit grounding (bottom).

If the dashed connection to the ideal-gear is missing the model would simulate, but generate incorrect results (if the torque-generator connection to ground is missing the model is singular). The intent of the restriction was to catch such cases early and generate good diagnostics.

However, when checking the Modelica Standard Library according to those semantics it was revealed that the situation was more complicated – in particular several Electrical Machine models had one Boolean parameter controlling *multiple* components including a connector, and in those cases leaving the connector unconnected was used and normal.

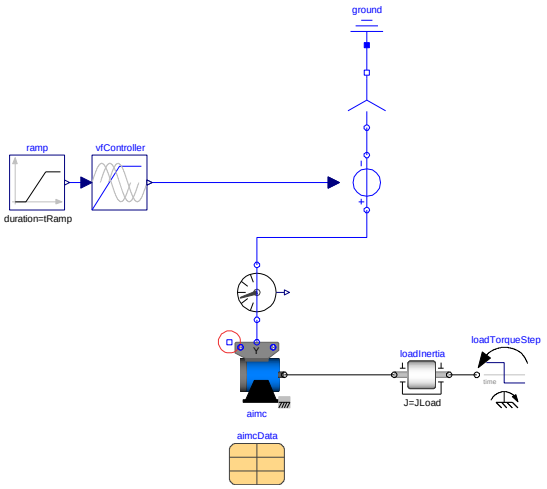

Figure 4 Machine model with unconnected conditional connector.

The red circle marks the single phase (“star”) connector of the terminalBox, it is only available in the Delta-configuration, but as shown here it is not always connected. On the other hand it was known that several models with unconditional connectors had assertions to ensure that they were connected (from the outside). Note, there are also some causal connectors with redundant

assertions to check that the connectors are connected, those assertions can just be removed – and predate the improvements in Modelica 3.0.

Thus MAP-Lang in Modelica 3.6 decided to replace the connection restriction based on whether the connection was conditional or not by an annotation indicating whether it must be connected, `mustBeConnected`, (and additionally one saying that it may only be connected once, `mayOnlyConnectOnce`). Both of them are given as a string indicating the reason – and for a conditional connector the restrictions are only active when it is conditionally active.

In this case the optional support flange could have:

```
Support support(
      phi=phi_support,
      tau=-flange.tau) if useSupport
    "Support/housing of component"
annotation (
  mustBeConnected="If the optional support
flange is enabled it must be connected",
Placement(transformation(extent=
  {{-10,-110},{10,-90}})));
```

This ensures that the previous correct examples work, and if the connect was missing a specific error message is given, e.g. in Dymola

The connector torque1.support was not connected from the outside, and it must be connected since: "If the optional support flange is enabled it must be connected"

The `mayOnlyConnectOnce` can be used in combination with automatic sizing to ensure that there is only one connection to each array element.

5.1 Implementation details

The connection restriction in Modelica 3.3 was not originally implemented in Dymola, and shows that even seemingly obvious improvements should be fully test-implemented before being added to the specification.

Note that one could think of multiple possible interpretations of “connected”: an active connect-statement involving that connector (used for `mustBeConnected`), or that its elements are part of a connection-set with additional elements (used for `mayOnlyConnectOnce` with specific restrictions for streams-connectors). The latter ensures that redundant connections are ignored, and the special rules for streams-connectors imply that sensor components are ignored, and thus one can, e.g., add a temperature-sensor without violating the restriction.

6 Multilingual support of Modelica

The documentation of the Modelica Standard Library is only written in English, whereas many tools support translation of their User Experience to different natural languages – in order to ease the use for non-English users.

Modelica 3.6 allows Modelica libraries (including the Modelica Standard Library) to provide translations without modifying the actual Modelica source code of the library; this was proposed and (test-)implemented by ESI. However, actually providing a localized User Experience requires both that the tool support using the translation and that the translation exists for the specific library – thus getting the benefit of this addition may take longer.

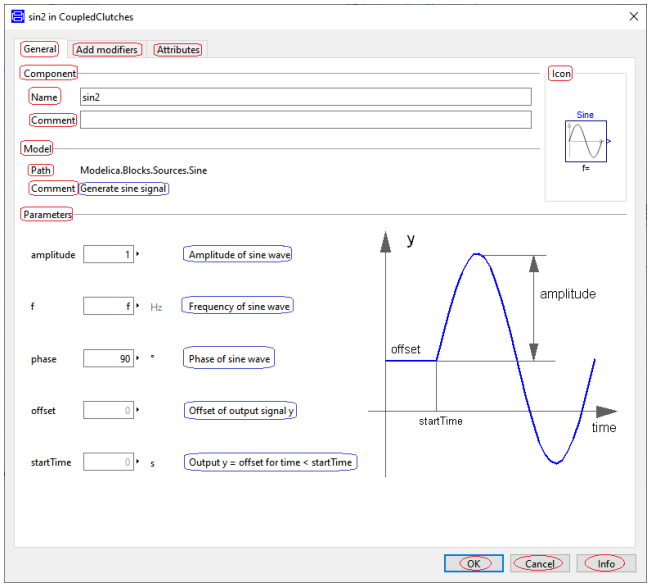

Figure 5 Dymola parameter dialog for Sine-block. Multilingual support means translating the texts in blue ovals. Texts in red ovals are already translated as part of tool settings (currently only for Japanese).

6.1 Implementation feedback

The multilingual support in Dymola 2023x Refresh 1 is only partial, but it revealed two important issues to consider.

The first is that a library maintainer for e.g., the Modelica Standard Library should update the English texts even if they normally work in another natural language. The simplest way to handle that is make it easy to disable the translations to keep the possibility of directly modifying the description and documentation in the library.

The second is that the descriptions often directly and indirectly reference the name of components, which makes translation more complicated. Note that the names of parameters and classes are deliberately not translated (and the Modelica language itself also uses English keywords).

Consider a parameter named “startTime”, with a description “Output y = offset for time < startTime”. This indicates two problems – first “offset” and “y” are other component names and should likely not be translated, and second that the description does not say that it is a “start time” (or “Time Sine Wave Starts”) since it is implied by its name. It is obviously possible to handle during the translation – but it means that it is not just a matter of merely directly translating the existing descriptions.

DOI
10.3384/ecp204367

Additionally the existing documentation often also uses images for showing how the model works, including the names of parameters. Thus the user should at least be able to recognize the English parameter names.

The partial support in Dymola is intended to give library authors the possibility to start providing the translation, and thus it is possible to generate the entire translation template – and use the translation of a few key impacting items like class and component descriptions.

7 Conclusions

This paper demonstrates that Modelica 3.6 has new powerful improvements. Dymola 2023x Refresh 1 and 3DEXPERIENCE 2023x FD03 supports these features (clearer parameter defaults, clearer start-value priority, connection restrictions, selective model extension, and multi-lingual support), and other tools have also released or are working on support for these features – and the goal of this paper is to improve portability by helping other implementers support these features. In particular, the selective model extensions considerations for deselecting connections and automatic connector sizing; and user experience for undefined modification and start-value precedence.

Acknowledgements

The work on Modelica 3.6 has been carried out by MAP-Lang. The author (who is also the chair of the group) acknowledges the hard work of the members of the group and the interesting discussions.

References

Bürger, Christoff (2019): Modelica language extensions for practical non-monotonic modelling: on the need for selective model extension. In: *Proceedings of 13th International Modelica Conference* 277-288 http://dx.doi.org/10.3384/ecp1915727

Casella, Francesco (2011): Selection of missing initial equations and of start attributes for alias variables URL: https://github.com/modelica/ModelicaSpecification /issues/561

Dassault Systèmes. (2023) Dymola 2023x Refresh 1: *Dymola, Dynamic Modeling Laboratory, User Manual.* Dassault Systèmes AB, Lund, Sweden.

Elmqvist, Hilding, and Martin Otter. 1994. Methods for Tearing Systems of Equations in Object-Oriented Modeling. Proceedings ESM'94, European Simulation Multiconference, Barcelona, Spain, June 1-3, pp.326--332.

Elmqvist, Hilding, Sven Erik Mattsson, and Christophe Chapuis (2009): Redundancies in Multibody Systems and Automatic Coupling of CATIA and Modelica. In: *Proceedings of 7th International Modelica Conference* 551-560 http://dx.doi.org/10.3384/ecp09430113

Franke, Rüdiger, Francesco Casella, Martin Otter, Michael Sielemann, Hilding Elmqvist, Sven Erik Mattsson, and Hans Olsson (2009): Stream Connectors – "An Extension of Modelica for Device-Oriented Modeling of Convective Transport Phenomena". In: *Proceedings of 7th International Modelica Conference* 108-121 http://dx.doi.org/10.3384/ecp09430078

ISO/IEC. (2020). ISO International Standard ISO/IEC 14882:2020(E) – Programming Language C++. Geneva, Switzerland: International Organization for Standardization (ISO). URL: https://isocpp.org/std/the-standard

MAP-Lang (2023): Modelica Language Specification. URL: https://github.com/modelica/ModelicaSpecification/

Miller, Bruce (2023): A LATEX to XML/HTML/MathML Converter URL: https://math.nist.gov/~BMiller/LaTeXML/

Olsson Hans, Martin Otter, Sven Erik Mattsson, and Hilding Elmqvist (2008): Balanced Models in Modelica 3.0 for Increased Model Quality. In: *Proceedings of 6th International Modelica Conference* 21-33 https://modelica.org/events/modelica2008/Proceedings/sessions/session1a3.pdf

Olsson, Hans (editor) (2023): Modelica - A Unified Object-Oriented Language for Systems Modeling Language Specification Version 3.6. URL: https://specification.modelica.org/maint/3.6/MLS.pdf