Design proposal of a standardized Base Modelica language

Gerd Kurzbach[1] Oliver Lenord[2] Hans Olsson[3] Martin Sjölund[4] Henrik Tidefelt[5]

[1]ESI Germany GmbH, Germany, gerd.kurzbach@esi-group.com>
[2]Robert Bosch GmbH, Germany, oliver.lenord@de.bosch.com
[3]Dassault Systèmes, Sweden, hans.olsson@3ds.com
[4]Department of Computer and Information Science (IDA), Linköping University, Sweden, martin.sjolund@liu.se
[5]Wolfram MathCore, Sweden, henrikt@wolfram.com

Abstract

This paper is presenting the design proposal of a simplified version of the Modelica language. Base Modelica is designed to serve as an intermediate representation enabling a clean separation of front-end and back-end matters when processing a Modelica model. Furthermore, it is designed to allow restructuring the Modelica Language Specification considering two parts: the basic features and the advanced language constructs.

After discussing the motivation, solution approach, and risks, the paper is highlighting a selection of design choices that have been made for the current pre-release version of the language. Code examples are given to illustrate and highlight various aspects of the language. Open issues, conclusions, and an outlook finalize the paper.

By attracting more tool vendors and researchers to work with this intermediate representation the whole Modelica community is expected to benefit from new utilities to inspect, analyze, optimize, and process equations-based models in general and Modelica models in particular.

Keywords: Modelica Language, intermediate representation, equation-based language, language design

1 Introduction

The Modelica language, published as v1.0 in September 1997, has been widely accepted as modeling language to describe the behavior of cyber-physical systems. Various tool vendors have developed and successfully marketed simulation environments including a Modelica kernel that is able to translate Modelica models describing mixed continuous-discrete differential algebraic-equation system (hybrid DAE) into highly efficient simulation code. From the very beginning, the development of tools has been accompanied by the development of model libraries covering a wide spectrum of physical domains and fields of application. The Modelica Language Specification and the Modelica Standard Library are maintained by the non-profit Modelica Association and are licensed under the open source 3-clause BSD License for the Modelica Association. An ecosystem of researchers, tool vendors, library developers and users has evolved over the years and is continuously growing.

The LLVM compiler infrastructure (Lattner and Adve 2004) has turned out to be a great success over the last twenty years. A central part of the design is the well specified LLVM intermediate representation, which has been a source of inspiration for also introducing a well specified intermediate format for Modelica translation and tool infrastructure. Earlier, the Java Virtual Machine architecture (Lindholm et al. 2023) has also proven intermediate languages to play a central role in the development of ecosystems around a language.

1.1 State of the art

The versatile modeling language Modelica is in particular well-suited to formulate multi-physics problems starting from first principles in an easily comprehensible textbook style. The combined textual and graphical representations provide very accessible views on component, subsystem and system level. A high level of reuse is enabled through the concept of acausal connectors. Variants can be managed in a convenient fashion through inheritance and modifications to avoid code duplication, which leads to a much-improved maintainability of model libraries and performing simulation studies of entire system families. The understandability of domain libraries and larger system models hinges on designing a proper model architecture, which requires a good understanding of Modelica's object-oriented programming model. High-quality libraries are available that demonstrate how to apply powerful object-oriented language constructs in a meaningful way, along with graphics, documentation, and other usability enhancements.

Modelica modeling tools support these concepts through integrated development environments (IDE) providing multiple views on the model to navigate through the instance hierarchy and apply modifications on different levels of the system structure. In addition, the output of a "so-called" *flattened model* is supported by common Modelica compilers. This textual output gives an unobscured view of the effective set of equations as they occur after instantiation and lowering of hierarchical models, applying replacements and other modifications. Typically, the flat Modelica output looks like Modelica code, but it is not a valid Modelica model that could be further processed by other tools. The flat Modelica output is not standardized and differs between tools and can even differ between tool versions.

For tool vendors the development of a Modelica compiler is a substantial investment, strongly dependent on the availability of highly educated and knowledgeable experts in compiler construction. Especially the concept of replaceable packages – heavily used in the media and thermo-fluid libraries – are very involved to get perfectly right. The OpenModelica project estimates 40 person-years spent in the project, with 15-20 person-years for a minimal simulation environment (excluding the graphical user interface). Approximately 7 person-years are estimated for the development of a Modelica front-end capable of parsing and lowering a Modelica model into a form that can be printed as flat Modelica code. This lowered model is the starting point for further model transformations, typically carried out by the so-called back-end, towards an optimized target-specific simulation code.

As of today, the Modelica Association is listing 10 Modelica simulation environments. Not all of these have their own Modelica kernel. The test coverage of available Modelica libraries differs between the Modelica kernels.

The latest version of the Modelica Specification (Modelica Association 2023) reads 300 pages (excluding the appendix) covering basic language concepts (operators, expressions, types, classes, arrays, functions, declarations, scoping, name lookup), hybrid DAE modeling-related aspects (equations, events, synchronous language elements, state machines) and object-oriented respectively component-oriented language elements (units, interfaces, connectors, connections, stream connectors, inheritance, overloaded operators, packages).

1.2 Problem statement

While Modelica is very powerful and easy to use, it is a complex language. The high complexity leads to challenges on different levels.

End-users (modeling and simulation experts) and Modelica library developers are confronted with:

- Compatibility issues between different Modelica tools in part due to inconsistent interpretations of the Modelica specification.
- Limited expressiveness of existing flat Modelica outputs for debugging unexpected behaviors, e.g., priority of conflicting start values, or clock partitioning.
- Lack of third-party utilities for operating on the flat Modelica output due to lack of standard.
- Lock-in effects due to third-party utilities getting tied to a specific Modelica tool.

Tool vendors are facing:

- High onboarding effort for new employees working on the Modelica translator to become productive due to interdependence of front-end and back-end matters.
- High entry barrier for non-Modelica tools to participate in the Modelica ecosystem.
- Difficulty to foresee and support all possible usages of the language.
- Lack of a common format to settle questions about the interpretation of the Modelica Language Specification with other tool vendors.

The design group of the Modelica Language Specification (MAP-Lang) is challenged by:

- Need to guarantee the consistency of a large and complex specification.
- Hard to integrate changes or enhancements due to many potential side effects to be considered.
- Difficulty to specify the semantics in an unambiguous way.
- Risk of language innovations creating high implementation efforts.

This leads to the situation that the objective of Modelica as a widely accepted, free and open modeling language is threatened:

- The entry barrier for new tool vendors is very high.
- The testing effort of Modelica libraries to guarantee compatibility across different tools is so high that only a limited number of tools are fully supported.
- It is difficult to use as foundation for other standards due the high complexity.
- A risk of vendor lock-in persists despite the commitment to Modelica as an open standard.
- Poor ability to respond quickly to innovations in competing modeling technologies.

1.3 Solution approach

We propose a standardized **Base Modelica** language that could become an integral part of the Modelica Specification. The translation of a (full) Modelica model would then be described as a two-step process, where high-level language constructs are first removed by *lowering* the model to Base Modelica, after which the Base Modelica semantics define the dynamic simulation behavior.

The Modelica and Base Modelica languages have a large overlap in the syntax of expressions, functions, equations, and algorithms. These parts should be defined in a common part of the Modelica Specification, where any differences in requirements or semantics between (full)

Modelica and Base Modelica are clearly marked. In addition to the common part, the high-level language constructs of Modelica would be described in one part, while lower-level constructs specific to Base Modelica are described in another.

This is similar to the approach being taken by System Modeling Language (SysML), which is being restructured to be an extension to the Kernel Modeling Language (KerML) instead of a UML profile (The Object Management Group 2023). The difference between Modelica and SysML is that SysML will add support for domain-specific applications through language extensions whereas these are still included in the (full) Modelica language.

1.4 Benefits

From the perspective of the MAP-Lang, the separation of basic language constructs (Base Modelica) from the more high-level constructions will facilitate more efficient work and rapid development of the two aspects of the Modelica language and generally improve the readability and maintainability of the specification. Examples of how high-level constructs are lowered to Base Modelica will help to avoid misinterpretations. Changes applicable to Base Modelica can be discussed and evaluated before deciding how to integrate them in Modelica. A working group with focus on the equation model and simulation semantics could play a very important role in future developments of new language features such as varying-structure systems, or integration with PDE solvers.

From a tool vendor perspective, organizing the development work of a Modelica tool will be easier thanks to a natural separation into front-end and back-end matters, with the front-end taking care of the lowering the Modelica model to Base Modelica and the back-end transforming the Base Modelica model into an executable form, e.g., a simulator. A standardized Base Modelica output will allow a much easier identification of compatibility issues between different tools. The much simplified Base Modelica language will provide an entry point for new tools to enter the Modelica ecosystem.

These types of new tools and services could be:

- Other high-level languages or modeling tools using Base Modelica as a target language, e.g., dedicated control engineering tools, or symbolic math packages.
- Advanced model transformation techniques applied on the equation level.
- Specialized tools providing advanced analysis (e.g., occurrence of algebraic loops, model-based fault detection and isolation) and/or visualizations of equation systems (e.g., bipartite graphs).
- Extraction and injection of equations to simplify or reduce the model for simulation speed-up.
- Platform for academic research on dynamic systems, e.g., symbolic or numeric methods.

In the context of the publicly funded ITEA3 project 15016 EMPHYSIS (Sep. 2017 – Feb. 2021), an early prototype for the exchange of equation-based models between Modelica and non-Modelica tools has been developed. Based on this prototype two use cases have been evaluated and documented as demonstrators (EMPHYSIS Consortium 2021, D7.3 and D7.4) to illustrate the benefit of having an equation-based representation in a model-based development workflow for embedded control and diagnosis functions.

The end-users and Modelica library developers will benefit from a improved portability of their models and libraries due to identified and resolved inconsistencies between Modelica implementations and Modelica Specification. This will allow a more flexible usage of the available tools. Comparison of different compiler back-ends will be possible. In combination with the obfuscation of Base Modelica outputs, sharing of IP-protected models will be simplified.

Furthermore, other model exchange standards could also benefit from a standardized Base Modelica. This is in accordance with the Equation Code proposed as an additional model representation within the eFMI container architecture (Lenord et al. 2021). This additional representation has been proposed for future versions of the eFMI standard aiming to share an acausal representation of the equation system that has served as the basis for the derived algorithmic and target-specific representations. The proposed Base Modelica with some additional restrictions could be directly referenced by the eFMI standard to specify the additional Equation Code model representation that would provide more flexibility and transparency in the generation of code for embedded applications.

1.5 Potential risks

However, we also recognize that there are potential risks that may reduce some of the benefits.

In particular:

- Modelica translators rely on heuristics for symbolic transformations based on the structure of the Modelica code. Some of this structure, e.g., start-value priority, has been given a standardized representation in Base Modelica. However there also exists structure that will require the use of vendor-specific annotations, leading to portability issues of the Base Modelica code. For example the alias elimination selection may depend on whether a variable is conditional.
- Base Modelica has not been designed with intermediate stages of symbolic transformations in mind. Thus its usefulness for representing those stages is not clear.
- Despite the significant effort behind the current state of the Base Modelica design work there is a consider-

able remaining investment in separating the Modelica Specification along these lines. This could detract from other evolution of the Modelica language.

Understanding the risks should make it easier to avoid the bad consequences.

2 Selected design choices

From the intended usages of Base Modelica the following design goals are derived:

- Simple enough to be attractive for applications that essentially just want a simple description of variables and equations, meaning that many of the complicated high level constructs of Modelica are removed.
- Expressive enough to allow the high level constructs of Modelica to be reduced to Base Modelica without loss of semantics.
- When Base Modelica serves as an intermediate representation of the translation of a higher level language (such as Modelica), errors detected in Base Modelica code shall be traceable to the original code.
- Human readable and writable, since not all usages assume Base Modelica being produced from a higher level language by a tool.

A selection of design choices to achieve these goals is presented in the following subsections.

2.1 Variable naming scheme

Identifiers in Base Modelica fall into three namespaces:

- Space of mangled class names and component references that allow mapping back to a hierarchically structured class tree and simulation result.
- Space of reserved names for current or future use in the Base Modelica specification.
- Space reserved for tools producing Base Modelica code, without mapping to names in a simulation result.

For example, the following Base Modelica parameter would appear as `const.k` in the simulation result:

```
parameter Real 'const.k' = 1.0;
```

In general, the mangling scheme is more involved than just wrapping in single quotes, but the details are omitted for brevity.

The mangled names are always quoted identifiers (*Q-IDENT* in the grammar), and since quoted identifiers are rarely used in Modelica code, it is often easy to guess whether a code fragment is Modelica or Base Modelica by just looking at the names of classes and components.

2.2 Simplified grammar

Base Modelica has a grammar which has been simplified in many ways compared to Modelica, by removing high-level language constructs. A clear sign of this is the many Modelica keywords that are not keywords in Base Modelica, including: **`block`**, **`class`**, `connect`, **`connector`**, **`constrainedby`**, `each`, **`expandable`**, **`extends`**, `final`, `flow`, **`import`**, **`inner`**, **`operator`**, **`outer`**, **`protected`**, **`public`**, **`redeclare`**, and `stream`. However, Base Modelica also comes with syntax for lowered constructs that do not exist in Modelica.

The top-level structure of a Base Modelica program is given by the following piece of grammar:

```
base-modelica :
  VERSION-HEADER
  package IDENT
    ( decoration? class-definition ";"
    | decoration? global-constant ";"
    )*
    decoration? model
        long-class-specifier ";"
    ( annotation-comment ";" )?
  end IDENT ";"
```

A very small Base Modelica model generated from full Modelica could look like this:

Listing 1. A minimal (non-empty) Base Modelica model.

```
//! base 0.1.0
package 'M'
model 'M'
  parameter Real 'const.k' = 1.0;
end 'M';
end 'M';
```

The mandatory Base Modelica version header comment is technically necessary to tell with certainty that this is a Base Modelica listing, and not a Modelica listing. It is included here for completeness, but is generally omitted in examples where it is clear from context or content that the language is Base Modelica.

A *class-definition* in Base Modelica can only be either a record definition, a function definition, or a short class definition, and cannot contain nested class definitions. Similarly, the model defined at the end cannot contain nested class definitions, meaning that all classes are defined in a flat structure under the top-level package (which shall have the same name as the model inside it).

The *decoration* is a source location decoration for use when the Base Modelica code has been generated from another source, such as a Modelica model.

A notable example of syntax added for describing lowered constructs is the modeling of clock partitions, see section 2.13.

2.3 Restricted modification

For any Base Modelica component declaration, modifications are required to be expressed in a way that avoids the need for conflict resolution and complicated merging strategies:

- Hierarchical names are not allowed in modifiers, meaning that all modifiers must use the nested form with just a single identifier at each level.

- At each level, all identifiers must be unique, so that conflicting modifications are trivially detected.

Lookup restrictions ensure that modifications in short class definitions, record definitions, and function definitions can only make use of constant expressions. Further, these modifications are not allowed to specify different values for different elements of an array. As a result, a named type in Base Modelica can be represented very compactly compared to a named type in Modelica.

When lowering a Modelica array component with a heterogeneous modification, the modification needs to be placed inside the Base Modelica `model` part, as model component declarations is the only place where heterogeneous modification is allowed.

Base Modelica does not have the `final` keyword to indicate that further modification is not allowed. For most uses of `final` in Modelica, this just means that violations of `final` must be detected during lowering. However, the special case of a final modification of the `start`-attribute also requires preventing that the `start`-attribute can be modified at the time of simulation initialization, and will be described in subsection 2.11.

2.4 No connect equations

Connect equations in Modelica play an important role to enable reuse of components and build-up a component hierarchy, as illustrated by the very simple example in Figure 1. The Modelica semantics already describes how to transform connect equations into basic mathematical equations. Hence, there is no need in Base Modelica to keep connect equations. Furthermore it would have been difficult to preserve the concept of connect equations, as the lowering process to Base Modelica is removing the hierarchy of components in terms of which the connect equations are defined. This is a significant language simplification compared to Modelica, especially when it comes to expandable connectors.

An example of how the previously mentioned train model is translated from Modelica, see Listing 2, to Base Modelica is shown in Listing 3

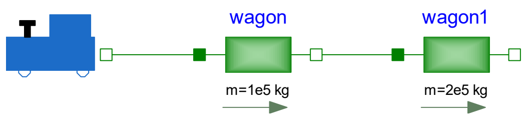

Figure 1. Diagram view of a Modelica model of a train. The `connect`-equations are represented graphically as lines between the components.

Listing 2. Shortened Modelica listing of train model.

```
model Train
  Locomotive locomotive;
  Modelica.Mechanics.Translational.
      Components.Mass wagon(L=50, m=1e5)
    annotation (Placement(transformation(
        extent={{-16,-12},{4,8}})));
  Modelica.Mechanics.Translational.
      Components.Mass wagon1(L=40, m=2e5);
equation
  connect(waggon.flange_b, wagon1.flange_a)
    annotation (Line(points
        ={{4,-2},{18,-2}}, color={0,127,0})
        );
  connect(locomotive.flange_b, wagon.
      flange_a);
end Train;
```

Listing 3. Train model lowered to Base Modelica.

```
//! base 0.1.0
package 'Train'
model 'Train'
  parameter Real 'wagon.m' = 100000.0 "Mass
       of the sliding mass";
  parameter Real 'wagon.L' = 50 "Length of
      component";
  parameter Real 'wagon1.m' = 200000.0 "
      Mass of the sliding mass";
  parameter Real 'wagon1.L' = 40 "Length of
       component";
  Real 'wagon.s' "Absolute position of
      center of component...";
  Real 'wagon.flange_a.s' "Absolute
      position of flange";
  Real 'wagon.flange_a.f' "Cut force
      directed into flange";
  Real 'wagon.flange_b.s' "Absolute
      position of flange";
  Real 'wagon.flange_b.f' "Cut force
      directed into flange";
  Real 'wagon1.s' "Absolute position of
      center of component...";
  parameter Real 'locomotive.mass.m';
  parameter Real 'locomotive.m' = 100000.0
      "Mass of the sliding mass";
initial equation
  'locomotive.mass.m' = 'locomotive.m';
equation
  'wagon.flange_a.s' = 'wagon.s'-'wagon.L'
      /2;
  'wagon.flange_b.s' = 'wagon.s'+'wagon.L'
      /2;
  ...
  // From connections:
  'locomotive.flange_b.f'+'wagon.flange_a.f
      ' = 0.0;
  'wagon.flange_a.s' = 'locomotive.flange_b
      .s';
  'wagon.flange_b.f'+'wagon1.flange_a.f' =
      0.0;
  'wagon1.flange_a.s' = 'wagon.flange_b.s';
end 'Train';
end 'Train';
```

2.5 No conditional components or deselection

Conditional components in Modelica allow a limited structural variation that complements the more flexible *re-*

placeable concept. Base Modelica does not have conditional components.

When lowering to Base Modelica we must thus evaluate the condition of a conditional component, and if the condition is false remove the component and any corresponding connections. The reason for this is that conditional components go together with the connection handling in Modelica. Additionally expressions (excluding arguments to connect) should be checked to ensure that they do not use any conditional component before lowering, since that check is not possible in Base Modelica.

Component and connect deselections introduced in Modelica 3.6 are handled similarly.

2.6 No evaluation of (Base Modelica) parameters

Constant and parameter variabilities are well separated in Base Modelica. Expressions that are deemed necessary to evaluate during Base Modelica translation are required to be constant, implying that lowering a Modelica model often involves evaluating parameters in order to comply with Base Modelica variability requirements. Further, Base Modelica semantics of constant components and expressions ensure that such expressions can be evaluated during translation when needed. In particular, a *pure constant* function concept is introduced to restrict the functions that can be used in constant expressions.

As an example of not having semantics relying on the ability to evaluate parameters during translation, a natural simplification compared to full Modelica is that a Base Modelica `if`-equation is required to have the same equation count in every branch. That is, whenever a Modelica `if`-equation has unbalanced branches, lowering of the equation must cause the `if`-equation conditions to be evaluated, so that branches breaking the balance can be eliminated.

2.7 Types are constant

Types created in Base Modelica can only hold constant properties. Since a constant property can always be evaluated in Base Modelica, this ensures that the internal representation of a type in a Base Modelica tool does not require the complexity of expressions and component references. This makes Base Modelica types much more similar to types found in other popular programming languages, compared to (full) Modelica types. It also significantly reduces the implementation effort for a pure Base Modelica tool compared to a full Modelica tool.

As an example of Base Modelica types being constant, each dimension of a Base Modelica array type has a size that is either *constant* or *flexible*, where the latter only indicates the absence of a constant expression for the size of an `Integer` dimension. Outside functions, component declarations may only specify constant array sizes. In an array equation, the array type must have constant sizes.

The `constsize`-expression allows expressing constant assertions on array dimensions. In the listing below, the OK assignment in `'h'` shows a way to assign the result of `'f'()` to `'z'`. The assignment below it is an error because `size('z')` has non-constant variability due to the flexible size of the first dimension.

Listing 4. Using the `constsize`-expression.

```
function 'f'
  output Real[:, 'MyEnumType', :] 'y';
  ...
end 'f';

function 'h'
protected
  Real[:, 'MyEnumType', 3] 'z';
algorithm
  'z' := constsize('f'(), :, size('z', 2),
      3); /* OK. */
  'z' := constsize('f'(), size('z')); /*
      Error. */
end 'h';
```

2.8 Variability-constrained types

Similar to Modelica, a record definition may have variability prefixes `parameter` or `constant` on the component declarations of the record members. In Base Modelica, such a type is denoted a *variability-constrained* type, and needs to obey additional rules that ensure more clarity in the semantics compared to Modelica. For example, a function component may not be of variability-constrained type, but is allowed to receive an argument of variability-constrained type. A model component of variability-constrained type is also allowed to be in solved position of an equation or assignment, in which case the variability-constrained members shall be disregarded as needed to match the type of the other side of the equation or assignment.

2.9 Short class definitions

Short class definitions in Base Modelica may not include array dimensions. Hence, all named types in Base Modelica are scalar types, and when a component has array dimensions, all array dimensions will be present at the component declaration.

It should be noted, however, that a record type may contain members of array type (where the sizes must be constant, as the component declarations are not inside a function):

```
record 'R'
  Real 'x'[3];
end 'R';
```

2.10 Array subscripting of general expressions

While Modelica only allows array subscripts as part of the component reference syntax, Base Modelica allows applying array subscripts to general expressions. The only requirement is that the array subscripts are applied to a parenthesized expression, for instance, `(x.y)[1, 2]` . The ambition is to introduce the new syntax for Modelica, thereby avoiding the need to introduce new expression syntax in Base Modelica, while also making the generally useful feature available on both language levels.

2.11 More explicit initialization

Model initialization is very explicit in Base Modelica compared to Modelica. A notable difference compared to Modelica is that both `fixed` and `final` in Modelica are modeled using more elementary mechanisms in Base Modelica.

A parameter with a declaration equation in Base Modelica corresponds to a non-final fixed parameter in Modelica, and it is required that the declaration equation is solved with respect to the parameter so that it is possible to override the equation during initialization. For variables of higher variability, fixed initialization in Modelica is lowered to an explicit initialization equation, and non-fixed initialization is lowered to an explicit representation of guess-values. For example, consider the following Modelica parameter:

```
  final parameter Real p = 4.2;
```

In Base Modelica, this can be represented as:

```
  parameter Real 'p';
initial equation
  'p' = 4.2;
```

To make handling of guess values explicit, there is an implicitly declared *guess value parameter* `guess('x')` to represent the guess value for `'x'`. The guess value parameter may be defined by an initial equation in case it should not be possible to override during initialization, or using a Base Modelica *parameter equation*. A parameter equation is a special construct that is only allowed for guess value parameters, and in addition to expressing that the guess value may be overridden during initialization, it allows the guess value to be conveniently located next to the declaration of the variable to which it belongs (without leaving the variable declaration section of the model):

```
Real 'x';
parameter equation guess('x') = 1.5;
Real 'y';
parameter equation guess('y') = 2.5;
```

Guess value prioritization is made explicit in the form of a special kind of initial equation:

```
initial equation
  prioritize('x', 2);
```

Further, **`when`**-equations impose no constraints on the initialization problem. Lowering a Modelica **`when`**-equation may therefore result in explicit equations in the **`initial equation`** section of the Base Modelica model.

One notable thing which is not explicit in Base Modelica is the selection of which guess values should come into play, or how. This requires an analysis of the initialization problem equation structure that goes beyond what the lowering of Modelica is expected to deliver.

2.12 Records and function default arguments

Function input components are allowed to have declaration equations, but these are ignored. Hence, Base Modelica functions cannot have function default arguments. In the same spirit, declaration equations in record types do not define defaults of an implicit record constructor function (as they do in Modelica). Instead, the declaration equations (which can only have constant expressions by design) only define default modifications when the type is used in component declarations, and then get meaning depending on what modifications mean for different kinds of component declarations. For example, a modification on a model component declaration is equivalent to an equation in the model, whereas a modification on a function local or output component declaration equation is used to give initial values for the evaluation of the function body. A modification on a function input component is ignored similar to declaration equations, as argument values must always be passed for all function inputs.

2.13 Explicit clock partitioning

The implicit clock partitioning carried out by tools for full Modelica is made explicit in Base Modelica. The equations solved in a clocked sub-partition are placed in a dedicated `subpartition` construct, and the variables being determined by the sub-partition can be determined by a simple inspection of the equations. Listing 5 shows an example of a Base Modelica model with clock partitions.

Listing 5. Explicit representation of clock partitions.

```
package 'M'
model 'M'
  Real 'x';
  Real 'baseVar', 'cVar1', 'cVar2', 'cVar3'
    ;
  Real 'mixedVar1';

equation
  der('x') = 1;

partition
  Clock 'myClock' = Clock(1);
  Clock _subClock0 =
    subSample('myClock', 2);
  Clock _subClock1 =
    superSample(subSample('myClock', 2), 8)
      ;

  subpartition (clock = 'myClock')
  equation
    'baseVar' = sample('x');

  subpartition (clock = _subClock0,
      solverMethod = "ImplicitEuler")
  equation
    der('cVar1') = noClock('baseVar');

  subpartition (clock = _subClock1)
  equation
    'cVar2' = noClock('baseVar');
    'cVar3' = noClock('cVar1');
  algorithm
    'mixedVar1' := 'cVar2' + 'cVar3';

partition
  Clock _baseClock0 = Clock(1.1);
  ...
end 'M';
end 'M';
```

2.14 Source locations

The Base Modelica grammar allows code to be decorated with source location information to enable reporting errors pointing back to another source (typically, a Modelica model) from which the Base Modelica was produced. Each decoration consists of the @ sign followed by an integer that references some external, tool-specific, table of source location details. As illustrated by the listing below, decorations can be attached to expressions as well as many other constructs.

Listing 6. Source location decorations.

```
package 'Decorations'
@101 model 'Decorations'
  @202 Real x(@203 min = 0.0 @204);
equation
  @301 if x > 0.5 then
    @302 w = 1;
  else
    6 + w @304 = atan2(1 @305, 1);
  end if;
algorithm
  @306 w := 1 + (2 @303) @304;
end 'Decorations';
end 'Decorations';
```

3 Base Modelica open issues

The design of the proposed Base Modelica language is an ongoing effort documented and discussed on GitHub under the Modelica Change Proposal (MCP) 0031. This paper is presenting the results after reaching the first milestone of a design proposal version 0.1 including resolutions of all collected issues considered crucial for a first complete Base Modelica language. The design proposal has been specified as a modified Modelica grammar file along with a separate textual description of the semantic differences between Base Modelica against Modelica. This design proposal has been developed by representatives from four different Modelica tool vendors and is considered mature enough for being implemented and tested by Modelica tools.

Based on forthcoming test implementations it will be possible to reveal and collect issues needing further attention. Some potential issues are in need of gaining experience with test implementations to pinpoint the problems in order to decide if there really are any.

The following issues are already known:

- Reject or add support for non-constant nominal-attribute.
- Handling of `ModelicaServices`.
- Handling of external functions.
- Reuse of common components.

4 Conclusions & Outlook

The selection of designs proposed in section 2 illustrate how the complexity of the Modelica language can be significantly reduced by removing keywords from the grammar related to higher level constructs, enforcing implicit declarations to be expressed more explicitly, and being generally more restrictive. All this leads to a language that is still expressive enough to capture the semantics of a Modelica model being lowered to Base Modelica, but much more accessible for non-Modelica tool vendors and researchers seeking a standardized form of equation-based mathematical models. This will enable new parties to participate and enrich the Modelica ecosystem with new applications and methods producing or consuming Base Modelica.

It is clearly outlined how Base Modelica is derived by lowering Modelica. This indicates that Base Modelica is consistent with existing Modelica tools and applicable as a standardized intermediate format. Only limited development effort is expected to enhance existing Modelica front-ends to produce a Base Modelica output and Modelica back-ends to consume it. This will improve the abilities of tool vendors and library developers to understand and eliminate incompatibilities between tools to the benefit of the entire Modelica community.

This paper is aiming to lay the foundation for future discussions within the Modelica community and with the MAP-Lang in order to collect feedback to further improve this design proposal.

In future work, we are aiming to work closely together with tool vendors to develop prototypes to generate as well as consume Base Modelica representations. These prototypes will be an important proof of concept to reveal shortcomings of the current design and to give realistic estimates of the development efforts to be expected.

If this evaluation phase can be concluded with positive feedback a revised definition of the Base Modelica language shall then be defined. Based on this version a change proposal to refactor the Modelica Language Specification, considering Base Modelica as an integral part, shall be worked out and submitted to the MAP-Lang.

In the long run, we are aiming to convince Modelica and non-Modelica tool vendors to embrace Base Modelica as a widely used standardized language for equation-based models for many more tools and other standards to build upon.

Acknowledgements

This work has been funded in the context of the ITEA3 project 15016 EMPHYSIS by the Swedish Vinnova under grant number 2017-05121 and the German Federal

Ministry of Education and Research (BMBF) under the grant number FKZ 01|S17023. It has also been funded in the context of the German PHyMoS project by the Federal Ministry for Economic Affairs and Climate Action (BMWK) under grant number FKZ 19|20022A. The authors would also like to thank Kai Werther, Michael Tiller, Francesco Casella, and others for their inputs to the many discussions.

References

EMPHYSIS Consortium (2021). *EMPHYSIS – D7.9 eFMI for physics-based ECU controllers*. Tech. rep. ITEA3. URL: https://itea4.org/project/workpackage/document/download/7675/D7.9_Public_DemonstratorSummary.pdf.

Lattner, Chris and Vikram Adve (2004-03). "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation". In: *CGO*. San Jose, CA, USA, pp. 75–88. DOI: 10.1109/CGO.2004.1281665.

Lenord, Oliver et al. (2021). "eFMI: An open standard for physical models in embedded software". In: *Proceedings of the 14th International Modelica Conference 2021*. Linköping, Sweden. DOI: 10.3384/ecp2118157.

Lindholm, Tim et al. (2023-03). *The Java Virtual Machine Specification, Java SE 20 Edition*. Tech. rep. ORACLE. URL: https://docs.oracle.com/javase/specs/jvms/se20/html/index.html.

Modelica Association (2023-03). *Modelica – A Unified Object-Oriented Language for Systems Modeling. Language Specification Version 3.6*. Tech. rep. Linköping: Modelica Association. URL: https://specification.modelica.org/maint/3.6/MLS.html.

The Object Management Group (2023). *OMG System Modeling Language version 2.0 Beta 1*. URL: https://www.omg.org/spec/SysML/2.0/Beta1.