Modelica language extensions for practical non-monotonic modelling: on the need for *selective* model extension

Christoff Bürger[1]

[1] Dassault Systèmes AB, Sweden, Christoff.BUeRGER@3ds.com

Abstract

A Modelica language extension for structural non-monotonic model variation is presented. It enables *selective* model extension: the well-defined refinement of models by deselecting components and connections not of interest or inappropriate for a new design. The need for such variations is explained by the example of Modelica Synchronous, whose adaptation is suffering from crosscutting synchronous decompositions that cannot be *anticipated* when continuous models are designed; instead, contradicting model structure has to be removed when an *actual* sampling is desired. Besides synchronous, further applications for selective model extension are investigated using our prototype implementation in Dymola.

Keywords: Modelica, model variation, synchronous

1 Introduction

Of key importance for Modelica is model variation support, enabling simulation of design alternatives and their step-wise refinement from idealistic prototypes to physically-detailed solutions. To that end, Modelica provides many different abstraction and variation techniques, like model extension, replaceable components, parameters and component modifications.

Having a strong heritage from object-oriented programming however, Modelica's model variation constructs are monotonic with respect to model structure because components, connections or equations can only be added but not removed when extending models. An unfortunately overlooked consequence of flatting is however, that such a structural-monotonic type-strictness, as known from class inheritance in traditional strongly typed object-oriented programming languages like Java or C++, is not required in Modelica. In Modelica, models are flattened before simulation. Flattening essentially reduces the design space of a set of models to a fixed number of instances according to a given parameterization and replaces the resulting instances with their corresponding fixed equation system (Modelica Association, 2017). The difference to traditional strongly typed object-oriented programming is striking: all instances are known before runtime, such that they can be statically constructed. There exists no *runtime* control-flow in Modelica that may cause different instantiations of entities; dynamic dispatch is not required, ultimately neglecting object-oriented polymorphism and the type-system restrictions that typically come with it (Wegner, 1987; Knudsen 1993)[1]. As a consequence, Modelica's current restriction that sub-models must inherit all components and connections of their base-models when extending – that model extension must be monotonic with respect to model structure – can be dropped.

Leveraging on this observation, the paper presents a new Modelica-language extension for non-monotonic modelling: *selective model extension*. Selective model extension can be used to exclude components and connections in a *well-defined* way from inheritance when extending models. Its semantic can be fully understood in terms of model-diagram edits, such that tools can support a convenient graphical user interface for structure-wise non-preserving model variation. The main contribution of selective model extension therefore is to enable unforeseen structural variability without requiring deliberately prepared base-models.

The paper starts with an evaluation on the need for non-monotonic model variation in Modelica (Section 2). To that end, the application of Modelica Synchronous (Elmqvist *et al*, 2012; Otter *et al*, 2012) to refine continuous models for discrete use-cases is chosen which requires non-monotonic modeling to handle the crosscutting clock-partitions of different synchronous designs. Based on the non-monotonic modeling requirements elaborated throughout that discussion, an exact syntax and semantic for selective model extension is presented (Section 3). A demonstration of general practical modelling-benefits, not only for Modelica Synchronous, follows (Section 4). A prototype implementation in Dymola is used on a sophisticated example taken from the Modelica Standard Library to show how selective model extension enables model-development along the lines of real engineering processes – i.e., in terms of step-wise model variation and adaptation – avoiding model variation inconsistencies and artificial intermediate models without physical meaning.

[1] Object-oriented languages typically require monotony of inheritance to ensure the functionality of entities is well-defined for all usage-contexts, independent of control-flows determining instantiation. If sub-classes could drop base-class functionality – i.e., inheritance could be non-monotonic – runtime errors are possible whenever base-class functionality is called on sub-class objects. Static type-systems enforce monotonic inheritance to avoid such errors in the first place.

2 Motivation: Modelica Synchronous adaptation challenges

This section motivates the need for non-monotonic model variation. As a practical problem the potential of Modelica Synchronous (Elmqvist *et al*, 2012; Otter *et al*, 2012) for *existing* examples of the Modelica Standard Library is investigated. The challenge is to enable use-case driven partial sampling of continuous systems without having to change them and with reasonable adaptation workload. As will be shown, the crosscutting of synchronous decompositions cannot be handled by monotonic model variation however; a refinement-based non-monotonic adaptation is required, giving rational for the *selective* model extension proposed in Section 3.

The modelling problems presented in the following are not Modelica Synchronous specific; they can be generalized as will be shown in Section 4.

2.1 Synchronous potential of the MSL

The Modelica Standard Library 3.2.2 has about 60 existing continuous *example* test-models *with controllers* whose discrete modelling might be of interest (cf. Appendix A for the used selection criteria). A lot of these test-models share the same controller, maybe differently parameterized. For example, the 35 models of `Electrical.PowerConverters.Examples` interesting for synchronous modelling share just five controllers defined in `PowerConverters.ACDC.Control` and `PowerConverters.DCDC.Control`. The remaining 25 test-models are much more heterogeneous however, making each a potentially worthwhile candidate for synchronous adaptation.

2.2 Objective: synchronous adaptation of continuous models via refinement

To adapt 60 test-models for synchronous is a major effort, in particular coordinating so many authors from different engineering domains. Involvement of the original authors therefore should be minimized and mostly only be required to ensure that the controllers of the existing continuous test-models are relevant for sampling from a domain perspective. After all, the existing test-models as such – their purely continuous modelling – are mature and useful.

To that end, sampling of their controllers should be an independent task, not requiring changing the original models. Instead, samplings should be introduced in terms of derived test-models that only add discrete partitions, i.e., by refinements adding samples, holds and clocks with respect to the components of an existing model. Such derived tests-models would be partially-discrete instances of their continuous originals, ultimately enabling validation and investigation of *different* samplings.

The original test-models would stay unchanged and cannot be corrupted by synchronous adaptation errors; their correctness is assured from previous model reviews and testing. Code duplication and inconsistencies are avoided and upcoming library changes eased. Ideally, future changes of a continuous model are either automatically incorporated in its derived partially-discrete models (in case the structural interface between continuous and discrete parts is not influenced, i.e., there are no new controller inputs or outputs), or result in translation errors of its derived partially-discrete models (denoting that the controller interface changed and samplings must be adapted).

To support such an iterative development process with seamless and incremental design from a continuous whole system model to different partially discrete variations via model-refinement is of uttermost importance for the success of Modelica Synchronous; it enables the incorporation and automatic change propagation of late continuous *and* discrete design changes and would be a distinctive Modelica feature compared to common block diagram based languages for *causal*-modelling of controllers.

2.3 Problem: monotony of model extension

To derive a partially-discrete model by sampling parts of an existing continuous model requires the introduction of samples, holds, clocks and their respective connections such that the derived model has a *consistent* clock partitioning. Modelica's existing model extension via **`extends`** is sufficient to add all required synchronization components. The derived model can also add the connections combining the sample and hold operators of the intended discrete model partitions with the model parts remaining continuous. The resulting derived model is *inconsistent* however, because it comprises all components of the original continuous model, particularly the old connections bypassing the sample- and hold-interface just introduced; clock-partitioning of the derived model fails due to the structural singularities resulting from having contradicting sampled and non-sampled connections. Since model extension via **`extends`** can only add components, modify the value of inherited components or exchange components deliberately prepared for variability via **`replaceable`**, it is impossible to fix clock-partitioning errors due to inherited connections and therefore *consistently* incorporate samplings.

2.4 Problem: prescient modelling

To enable sampling via model extension, parametrization or modification requires *deliberate preparations* of model parts that might become subject to sampling. For example, models could be prepared for sampling by pushing the parts constituting controllers into separate models and referencing them

DOI 10.3384/ecp19157277

as replaceable components well-suited for modification or by using conditional declarations instantiating a continuous or discrete design depending on parameterization. Such workarounds are in conflict with our objectives however, as they anticipate *specific* samplings before their actual need, implying changes of the original model when the need for a new specific sampling *actually* arises. Deliberate preparations of models to enable future samplings naturally only enable the prepared samplings, i.e., specific discrete use-cases. To anticipate all possible samplings of continuous model parts that might be of interest in future discrete use-cases is impractical however.

2.5 Problem: crosscutting synchronous decompositions

The design of controllers significantly varies depending on available sensor information (varying control input signals) and control-task splitting (varying model parts constituting controllers, for example due to independent asynchronous control vs. synchronous cascade-control with different sub- or super-samplings). Typically, many reasonable synchronous designs exist, each resulting in a specific clock partitioning. The clock partitions of *different* synchronous designs are likely in conflict however.

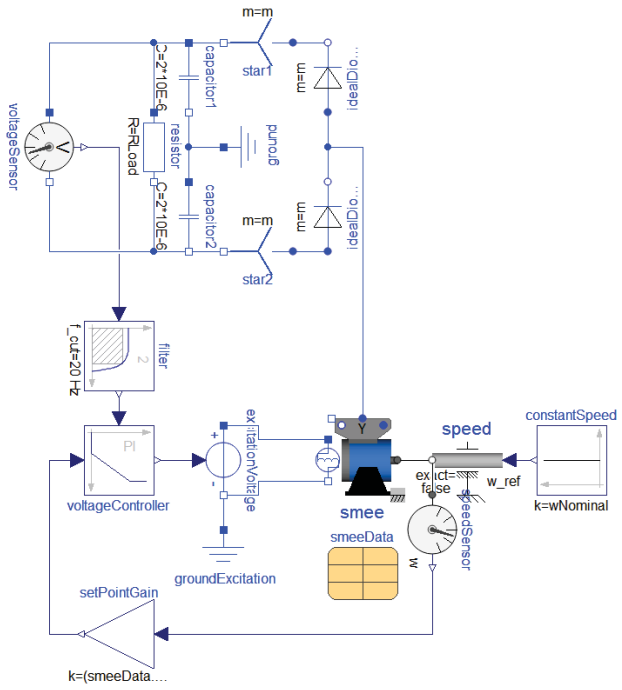


Figure 1. Induction machine with voltage controller.

Consider for example the electrical excited synchronous induction machine of the Modelica Standard Library presented in Figure 1 (`Electrical.Machines.Examples.SynchronousInductionMachines.SMEE_Rectifier`). Five different synchronous designs immediately come to mind for its voltage controller: (1) a fat controller, comprising not only the gain and PI controller but also filter, (2) a design with the filter being independent, either as (2.1) a separate asynchronous sampled system or (2.2) not sampled at all and (3) a cascade control, with the filter being (3.1) sub-sampled, in case set point changes are more critical than filtering the current voltage, or (3.2) super-sampled, in case the filter implementation requires higher sample rates than the rest of the controller. The clock partitions of all five variants are in mutual conflict although each, in itself, is sound.

Important for our investigation is that Modelica models already have a dominant decomposition with respect to their component hierarchy (network of interconnected hierarchical components); and it is that very hierarchy in whose terms model variation is defined using parametrizations, modifications and re-declarations, whereas model extension always preserves it. Clock partitioning however is about decomposing a model according to its differently clocked parts. Thus, even if a model's structure is aligned with *some* future synchronous design, it will be in conflict with *other* designs. Clock partitioning crosscuts the natural composition of physical systems as hierarchical component networks[2].

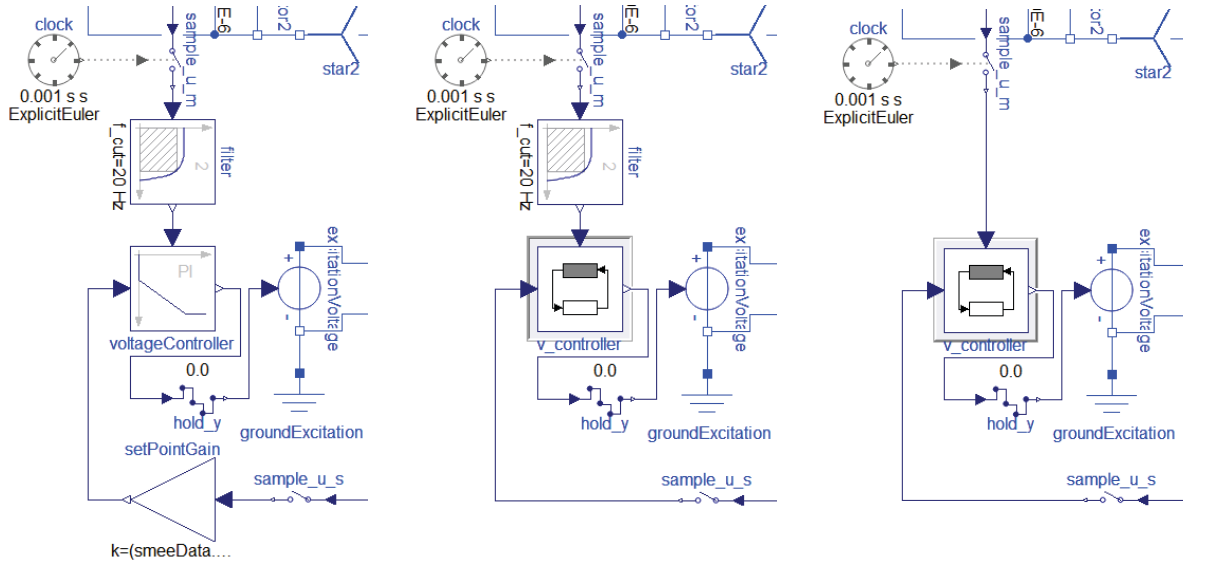
2.6 Solution: non-monotonic extension

To incorporate a specific sampling into an existing model means to modify its component network according to the sampling's crosscut, i.e., to change the model's *structure* at the intersection points of clock partitions and further control-design adaptations. Intersection points of clock partitions correspond to connections that must be removed; instead respective samples and holds are added, connecting the clock partitions. Control-design adaptations usually correspond to components that have to be removed because the new control-design is structural different duo to changed sensor and actuator usage (for example the filter and gain of Figure 1 may not be required by a third party library controller). All such changes are well-defined by removing connections and components that are superfluous and replaced by the intended synchronous design. The required refinement can be defined as ordinary model extension with parts of the original model excluded from inheritance, ultimately enabling structural non-monotonic changes.

3 Selective model extension proposal

This section presents a concrete proposal to enable non-monotonic modelling in Modelica. The proposed selective model extension enables the deselection of

[2]Implementation techniques for system parts cross-cutting a dominant component hierarchy are subject of aspect-oriented programming (Kiczales *et al*, 1997; Tarr 1999). Particularly object-oriented programming language extensions enabling crosscutting implementation are well-investigated. The proposed selective model extension can be seen in that tradition; it is Modelica-specific however, since it depends on static instantiation via model-flattening as explained in Section 1.



(a) simple sampling **(b)** off-the-shelf controller **(c)** off-the-shelf dampening-controller

Figure 2. Three control scenarios for the induction machine of Figure 1 (controller only excerpts).

connections and components when extending models. To start with, a simple sampling example sketches the new language concepts. The definition of actual syntax and semantic follows.

3.1 Selective model extension example

Consider again the controlled induction machine presented in Figure 1. Figure 2 presents three different synchronous control designs for it, implemented in the following using selective model extension to handle their structural non-monotonic variation.

(a) Simple sampling scenario: A straightforward synchronous control design is to just sample the control components of the original example. To that end, the connections between `voltageSensor` and `filter`, `speedSensor` and `setPointGain` and `voltageController` and `excitationVoltage` have to be replaced with likewise-connected samples and holds. Using selective model extension, the implementation of Figure 2 (a) is:

```
model SMEE_Rectifier_Sampled
  extends SMEE_Rectifier;
/* PART 1: Drop "outdated" continuous parts. */
  // Exclude connections from inheritance:
  for each extends
    break connect(voltageSensor.v, filter.u);
    break connect(speedSensor.w,
                  setPointGain.u);
    break connect(voltageController.y,
                  excitationVoltage.v);
  end for each extends;
/* PART 2: Introduce sampling. */
  // Introduce clock, samples and hold and…
  PeriodicRealClock clock(
    period = 0.001,
    useSolver = true);
  SampleClocked sample_u_m;
  Sample sample_u_s;
  Hold hold_y;
equation
  // …connect them:
  connect(clock.y, sample_u_m.clock);
  connect(voltageSensor.v, sample_u_m.u);
  connect(sample_u_m.y, filter.u);
  connect(speedSensor.w, sample_u_s.u);
  connect(sample_u_s.y, setPointGain.u);
  connect(voltageController.y, hold_y.u);
  connect(hold_y.y, excitationVoltage.v);
end SMEE_Rectifier_Sampled;
```

The new sampling-related components – the clock, sample and hold and their connections with inherited components – are introduced as used to and are subject to normal Modelica 3.4 semantic (Part 2). Also syntax and semantic of the `extends` clause are as used to, except that the `for each extends` block modifies the set of features the `extends` clause defines to be inherited (Part 1). Each `break connect` clause within a `for each extends` block removes the respective connection from the set of features the model inherits. Note the plural form *clauses*, implying that `for each extends` modifies *all* `extends` clauses of a model. In our case, just the extension from `SMEE_Rectifier` is modified, excluding the ingoing connections of `filter` and `setPointGain` and the outgoing connection of `voltageController` from inheritance; the deselections are `break connect(voltageSensor.v,filter.u)`, `break connect(speedSensor.w,setPointGain.u)` and `break connect(voltageController.y,excitationVoltage.v)`.

(b) Off-the-shelf controller scenario: Another reasonable design is to use an off-the-shelf controller provided by a specialized library, as shown in Figure 2 (b). To that end, the original control *components* have to be replaced. Note the plural; not a single `replaceable` component is changed, but the complete component network constituting the controller. Assuming the new controller still requires the filtering of voltage, only `voltageController` and `setPointGain` have to be removed. Using a selective model extension of scenario (a), the implementation of Figure 2 (b) is:

DOI 10.3384/ecp19157277

```
model SMEE_Rectifier_ExternalController
  extends SMEE_Rectifier_Sampled;
// Remove original controller with connections:
  for each extends
    break voltageController;
    break setPointGain;
  end for each extends;
// Introduce the new controller…
  replaceable ExternalController v_controller;
equation
// …and connect it:
  connect(filter.y, v_controller.u_m);
  connect(v_controller.y, hold_y.u);
  connect(sample_u_s.y, v_controller.u_s);
end SMEE_Rectifier_ExternalController;
```

The original controller and gain are excluded from inheritance via `break voltageController` and `break setPointGain`. Deselecting a component automatically deselects all its connections. Thus, the only thing to do besides deselecting the original control components is to integrate the new controller reusing the sampling inherited from scenario (a).

(c) Off-the-shelf dampening-controller scenario: Finally, an off-the-shelf controller with specialized dampening of its input voltage can be used as shown in Figure 2 (c). In that case, the original filter is not required. The implementation based on scenario (b) is:

```
model SMEE_Rectifier_DampeningExternalController
  extends SMEE_Rectifier_ExternalController(
    redeclare DampeningController v_controller);
  for each extends
    break filter;
  end for each extends;
equation
  connect(sample_u_m.y, v_controller.u_m);
end SMEE_Rectifier_DampeningExternalController;
```

Conclusions: Scenarios (a) to (c) demonstrated the consecutive synchronous adaptation of a continuous model. Each refinement step required structural non-monotonic changes in terms of removing superfluous connections and components inappropriate for a more sophisticated control design. Using selective model extension, the respective synchronous adaptations are possible without changing the original continuous model, ensuring configuration consistency of the controlled system when comparing the synchronous designs one another. Also diagrammatic consistency is improved; after all, the diagrams of Figure 2 are derived from Figure 1 by normal `extends` semantic and our Dymola implementation of deselections.

3.2 Syntax: selective extension clauses and inheritance modifications

Selective model extension as presented in Section 3.1 requires rule-additions to Modelica's context-free grammar. The changes required are very limited however. Only an additional alternative for `element` (cf. Appendix "B.2 Grammar" of the Modelica 3.4 specification) has to be added:

```
element :
  import-clause |
  extends-clause |
  selective-extension-clause | // new
  [ redeclare ]
  [ final ]
  [ inner ] [ outer ]
  ( (class-definition | component-clause) |
    replaceable (
      class-definition | component-clause)
    [constraining-clause comment])
```

whereas `selective-extension-clause` is:

```
selective-extension-clause :
  for each extends
  { inheritance-modification ";" }
  end for each extends
```

and `inheritance-modification` is:

```
inheritance-modification :
  break connect-clause | // Connection and…
  break IDENT // …component deselection.
```

with `connect-clause` and `IDENT` already well-defined in the specification. No new keywords are introduced. The new context-free derivations `for each extends`, `break connect` and `break IDENT` are syntax errors in current Modelica. As a consequence, the proposed selective model extension never changes the semantic of existing valid Modelica 3.4 models. Models that are syntactically invalid could theoretically become valid however, but chances are extremely low[3].

3.3 Semantic: terminology, well-formedness and interpretation

An important criterion of selective model extension is to ensure applications are meaningful. A selective extension is meaningful when *all* its modifications of the set of inherited elements are unambiguous and applicable, in which case it is called well-formed. Selective extensions that are not well-formed are modelling errors; they are meaningless, i.e., without unique interpretation defining the result of their application. The rest of this section defines well-formedness and interpretation for the proposed syntax.

3.3.1 Terminology

To ease further discussion, we define the following terms (words embraced by parenthesis are optional, only improving readability; the term "if X is evident" denotes "if X is already well-defined by context (i.e., specific) or not of particular interest (i.e., generic)"; the term "derivation" denotes a context-free derivation according to the syntax specified in Section 3.2):

(a) Context of selective extensions: A `selective-extension-clause` derivation S within a model A with arbitrary many `extends` clauses E_1, … E_n, that extend

[3]It is not likely that a syntax error happens to satisfy the proposed syntax. It is even less likely the respective model will further satisfy the semantic constraints explained in Section 3.3.2; and, due to deselections, it is close to impossible that it is valid considering existing Modelica well-formedness constraints like *"referenced components must be declared"* and *"the system of equations must be well-defined"*.

models $M_1, \ldots M_n$ respectively, is called a "selective (model) extension of $M_1, \ldots M_n$"; if $M_1, \ldots M_n$ are evident, just "selective (model) extension". $E_1, \ldots E_n$ are called "(local) extends-clauses of A". S is called "extends-modification of A". If A is evident, we just speak of "local extends-clauses" and "extends-modification". We say "S is local to A", "$E_1, \ldots E_n$ are local to A", "$E_1, \ldots E_n$ are local to S" and vice-versa; and we call an "α local to β" a "(local) α of β" and vice-versa. We further say "S modifies $E_1, \ldots E_n$" and "A selectively extends $M_1, \ldots M_n$ with respect to S". If $M_1, \ldots M_n$ or S are evident, we just say "A selectively extends". This terms are called "context of S"; if S is evident, just "selective extension context".

(b) Context of deselections: A selective extension S is a block; its body consists of the `inheritance-modification` derivations $m_1, \ldots m_n$ applied throughout the derivation of S. Each `inheritance-modification` derivation is called a "deselection of S"; if S is evident, just "deselection". The set $m_1, \ldots m_n$ are the "deselections of S"; if S is evident, just "deselections".

We distinguish two types of deselection:

(b.1) **`break`** `connect-clause` derivations are called "connection deselection"

(b.2) **`break`** **`IDENT`** derivations are called "component deselection"

If a deselection type is evident, we just say "element" instead of connection or component.

The subset of connection deselections of the deselections of S are called "connection deselections of S"; if S is evident, just "connection deselections". The deselections of S that are not connection deselections are called "component deselections of S"; if S is evident, just "component deselections".

The relations defined for S in (a) – like extends-clauses, extends-modification, modifies etc. – also hold for the deselections of S. We therefore can speak of the context of a deselection, defined by the model it is local to and the local extends clauses it modifies; and it is true by definition that *"deselections are extends-modifications of their local model and models selectively extend with respect to their deselections"*.

(c) Extent of selective extensions: Let $I_{extends}$ be the set of elements the local extends clauses of a model A define to be inherited; let connection elements be represented by their respective `connect-clause` derivations in $I_{extends}$ and components by **`IDENT`** derivations, i.e., their name. We call $I_{extends}$ the "preselective-extent of A". Let $D_{connection}$ be the set of connection deselections of A and $D_{component}$ the set of component deselections. We call two connections **`connect`**`(a1, b1)` and **`connect`**`(a2, b2)` "matching" if either `a1 = a2 ∧ b1 = b2` or `a1 = b2 ∧ b1 = a2`.

We call an inherited connection $c_i \in I_{extends}$ "extent of a (connection) deselection" $d =$ **`break`** $c_d \in D_{connection}$ if c_d and c_i are matching and say "c_i is deselected due to d" and "d deselects c_i"; if d is evident we just say "c_i is deselected", and if c_i is evident we just speak of a "deselected T (connection)" whereas T is the connector type of c_i. If also T is evident, we just speak of a "deselected connection".

We call an inherited component $c_i \in I_{extends}$ "extent of a (component) deselection" $d =$ **`break`** $c_d \in D_{component}$ if $c_d = c_i$ and say "c_i is deselected due to d" and "d deselects c_i"; if d is evident we just say "c_i is deselected", and if c_i is evident we just speak of a "deselected T (component)" whereas T is the component type of c_i. If also T is evident, we just speak of a "deselected component".

Let c_i be a component deselected due to a deselection d. We call the set $D = c_i \cup \{c \in I_{extends} \mid c \text{ is connection of } c_i\}$ the "transitive-extent of d"; if d is evident, we just speak of a "transitive-extent". For each $c \in D \setminus c_i$ we say "c is indirectly-deselected due to d"; if d is evident, we just say "c is indirectly-deselected" and, if c is also evident, we speak of an "indirectly-deselected connection". Indirectly-deselected connections are deselected connections. The transitive-extent of a connection deselection is just its extent.

We call the union of the transitive-extents of the deselections of A the "deselective-extent of A". Let $I_{deselected}$ be the deselective-extent of A; we call the set $I_{selected} = I_{extends} \setminus I_{deselected}$ "selective-extent of A"; if A is evident, we just speak of "preselective-", "deselective-" and "selective-extent".

Colloquial usage: Whenever we emphasize the act of *modelling* via introducing deselections for an element or element type E, we use the term "deselection of E" or "deselecting E"; if E is evident, we just speak of "deselecting". Likewise, we speak of "selection of E" and "selecting E" for removing, or deliberately not introducing, deselections for E.

3.3.2 Well-formedness

Five well-formedness constraints are proposed for selective model extension. The following list also gives a short rational for each constraint:

Constraint (1) Selective model extensions must be element of a model or block (i.e., the enclosing scope of a `selective-extension-clause` must be a `class-definition` whose `class-prefixes` are derived to **`model`**, **`block`**, **`partial model`**, or **`partial block`**; cf. Appendix B.2.2 of the Modelica 3.4 specification).

Rational: Connector classes are prohibited to use selective extension because their whole purpose is to define common interfaces; deselection of connector-components would essentially make the derived connectors incompatible. Types are excluded for similar reasons. Records are excluded to avoid runtime errors due to instances queried for deselected fields. Packages are excluded because they are used to define a modelling environment with well-defined features; to reduce availability of provided features contradicts

DOI 10.3384/ecp19157277

their purpose in the first place. Deselection of function in- and out-puts is prohibited, because call-sites depend on the applicability of a function's interface.

Constraint (2) The extent of deselections is not empty (i.e., for each deselection exists a local extends clause that inherits the deselected element).

Rational: Selective extensions should be meaningful, i.e., each of their deselections should be applicable. The constraint also makes it impossible to deselect beforehand, eliminating the risk that sub-models accidentally miss future base-model improvements.

Constraint (3) Deselected components are not modified by local extends-clauses.

Rational: Modifying a component and deselecting it within the same model hints at a modelling error.

Constraint (4) Deselected elements are not `final`.[4]

Rational: `final` is deliberately introduced by developers to prevent common model-configuration errors due to further modifications; this naturally encompasses modifications changing the *existence* of `final` modified components. The constraint also prevents the reintroduction of deselected `final` components as non-finals.

Constraint (5) Models have at most a single selective extension clause.

Rational: Since selective extension modifies all local extends-clauses, it makes sense to collect all deselections of a model within a single `for each extends` block. Doing so avoids scattering of inheritance modifications, ultimately increasing readability of models.

These constraints can be checked just considering the set of connections and components inherited due to *local* extends-clauses; there is no need to mutually compare the individual sets. Details, how inherited elements are defined, particularly if base-models apply other selective extensions, are not required.

Note that the proposed well-formedness constrains do not prohibit extending models from reintroducing components deselected. This is useful to solve currently non-manageable multiple-inheritance conflicts due to structural differences of base-models.

No further restrictions regarding the well-formedness of equations are proposed. Deselection of connections can result in structural non-singular equation systems however; likewise deselected components may result in base-model equations with unresolved references. The fallback on default equation well-formedness is important. It ensures the context of selective extensions is sound; in practice, this means that the "structural-holes" due to deselections must be properly fixed and invalidating base-model changes are caught. Note that speaking of "structural-holes" is reasonable, considering deselections are defined with respect to diagram-wise clearly distinguishable model parts; selective model extension is about the removal of interconnected component networks. To accidentally change model semantic not visible within the diagram layer, like non-connection equations, is impossible. As a consequence, deselection can be realized as graphical edit-operations in the diagram layer of Modelica tools.

3.3.3 Interpretation

Given the terminology of Section 3.3.1 and the well-formedness constraints of Section 3.3.2, defining an interpretation for well-formed selective model extensions is straightforward.

The objective of a selective extension is to exclude elements from inheritance. To that end, one has to take care of – colloquial speaking – three kinds of inherited elements: (1) the elements inherited by a model's local `extends` clauses (i.e., inheritance as used to from Modelica 3.4), (2) the elements excluded from this set and (3) the resulting actually inherited elements. With respect to Section 3.3.1, these sets are the preselective-, deselective- and selective-extent.

The extent definitions of Section 3.3.1 are constructive; first, the preselective-extent is derived, based on it the deselective-extent, finally followed by the selective-extent. Note that, the extents can be empty for the definitions to hold. The deselective-extent of a model without a selective extension is the empty set; such a model's selective-extent just is its preselective. This characteristic significantly limits the changes required in the existing Modelica specification to incorporate selective model extension.

In the end, the interpretation of selective model extension just boils down to the addition of the terminology introduced in Section 3.3.1 and a single modification of the Modelica 3.4 specification regarding the definition of inherited elements; the new definition is: *"the inherited elements of a model are its selective-extent"*.

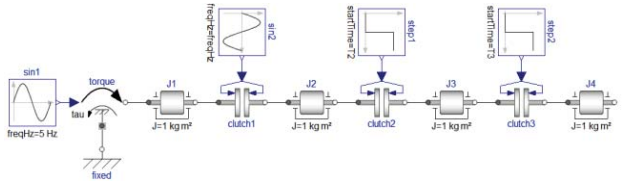
4 Advanced application scenarios

The applications of selective model extension presented so far are all in the domain of Modelica Synchronous. In the following, further applications, with the focus on general advantages for modelling from an engineering perspective, are investigated. To that end, selective extension is used to redesign an existing example scenario of the Modelica Standard Library; doing so will reveal implementation-shortcomings of the example and how non-monotonic modeling can be used to avoid them. First however, another important use case for selective model extension is presented: to adapt whole system models for further external or component usage.

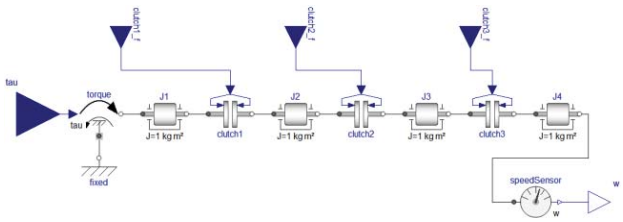
[4] Constraint (4) does *not* prohibit the deselection of components containing `final` elements, as long as the deselected component itself is not `final`.

4.1 Component extraction example

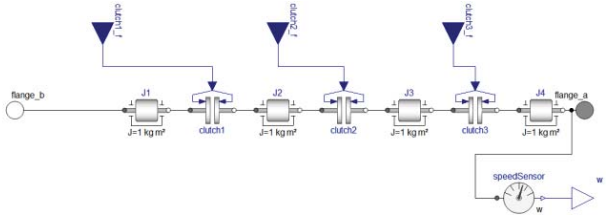
Figure 3 presents the well-known coupled clutches example of the Modelica Standard Library (`Mechanics.Rotational.Examples.CoupledClutches`) and two variations of it. The first variant prepares its export as Functional Mockup Unit (FMU) that can be distributed to third parties for simulation in non-Modelica contexts (Modelica Association, 2014); the second variant prepares the coupled clutches for usage as component within further Modelica models. In both cases, the fixed input stimuli of the original example must be removed and replaced by respective input connectors. The normal forces of the clutches just become real inputs; the in- and outputs of the inertias depend on usage however.



(a) original coupled clutches



(b) extracted FMU model based on (a)



(c) extracted component model based on (b)

Figure 3. Coupled clutches FMU/component extraction.

For FMU simulation, the input `tau` for the first inertia is a torque and the output of the simulation is the absolute angular velocity `w` of the fourth inertia `J4`; thus, both are just real values. For component usage however, one would like to stay with the flange interface of the Modelica Standard Library for the in- and output of the first and last inertias. Doing so ensures proper flow-derivation of the cut-torques[5].

[5]Because flow-variables – and therefore Modelica-like automatic flow-value derivation – are not supported in the FMI 2.0 standard, users of the FMU-component of Figure 3 (b) have to be careful that torques are correctly modeled in application contexts of the FMU; in that sense the FMU-component is less flexible compared to the Modelica-component of Figure 3 (c). On the other hand, the flow-variables of the flange-interface are the reason why the Modelica-component is unsuitable for FMU-export and usage in external simulations.

Both adaptations are straightforward using selective extensions. For FMU extraction the implementation is

```
extends …Rotational.Examples.CoupledClutches;
for each extends
  break sin1;break sin2;break step1;break step2;
end for each extends;
```

accompanied by introducing and connecting the in- and output normal forces, torque and velocity as shown in Figure 3 (b). When extracting a component however, the torque adapter becomes superfluous since a proper flange input will be provided. In terms of the FMU model, the respective selective extension is

```
extends CoupledClutches_FMU;
for each extends
  break torque; break fixed; break tau;
end for each extends;
```

this time accompanied by introducing and connecting the in- and output flanges as shown in Figure 3 (c).

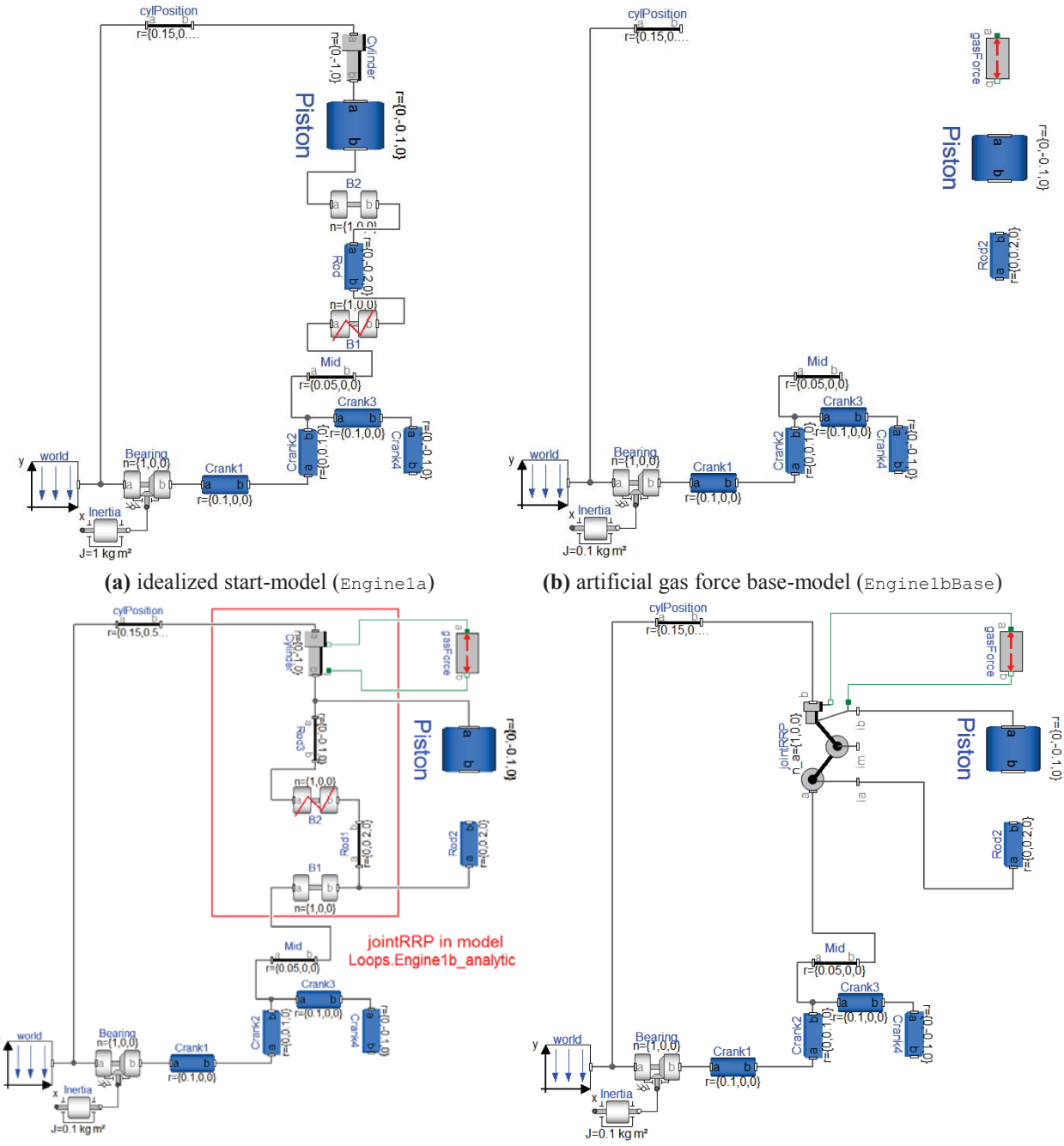
In conclusion, selective model extension enables to extract a component model from a whole system model and incorporate the usage-interfaces of future application contexts. Doing so we *know* the extracted component is working; after all it comes from a well-tested, whole system model with proper simulation that has just been lifted to a component on demand.

4.2 Domain-driven refinement example

Our final selective extension scenario is the step-wise design of a one cylinder engine, as exemplified by the `Engine1a`, `Engine1b` and `Engine1b_analytic` models in package `Mechanics.MultiBody.Examples.Loops` of the Modelica Standard Library. The basic idea is to design a final analytic engine model starting from an idealized model via one considering the gas force in the cylinder. Figure 4 summarizes the current solution of the standard library. There are several problems with it, all due to the lack of non-monotonic modeling means.

The most obvious inconsistency is that the models incorporating the cylinder's gas force (`Engine1b` and `Engine1b_analytic`) do not inherit from the idealized base model (`Engine1a`), but from a completely independent new `partial` model (`Engine1bBase`). The reason can be only understood by an investigation *starting* from the *final* analytic model: it introduces the `jointRPP` component, which encapsulates an analytic solution for original engine components. Thus, the final solution cannot extend the idealized start-design because it has to replace parts of the start-design's component network with something whose incremental design *is* the actual task. `Engine1bBase` was introduced to consistently configure *at least* the common components of models considering the gas force of the cylinder. But `Engine1bBase` is completely artificial: it cannot be simulated, its components are hanging in the air and it has nothing in common with the idealized model that was the original starting point for designing the engine. Quiet contrary it is the result of an inversed engineering process, taunting the natural design order.

DOI 10.3384/ecp19157277



(a) idealized start-model (`Engine1a`)

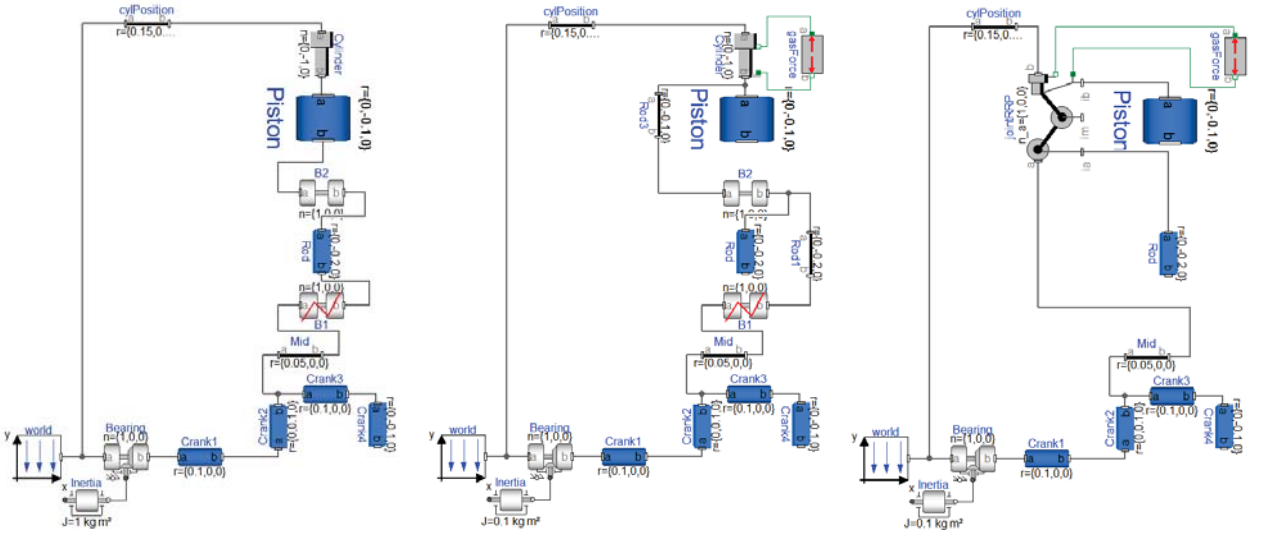
(b) artificial gas force base-model (`Engine1bBase`)

(c) gas force intermediate-model (`Engine1b`)

(d) analytic final-model (`Engine1b_analytic`)

Figure 4. One cylinder engine scenario (current standard library solution).

Since `Engine1bBase` does not extend `Engine1a` – in fact cannot – the obvious question is if the idealized and gas force incorporating models are *at least* consistently configured. This is an important issue because `Engine1bBase` is a partial model-copy of `Engine1a`; it is not obvious if differences are intentional or just copy-and-paste errors that slipped in throughout revisions. As it turns out there is a plethora of configuration differences however. First, the inertias are configured differently; likewise `r` of `cylPosition` is inconsistent. Second, the `a`-connector of the piston is connected with `b` of the cylinder in `Engine1a` but with `Rod3.a` in `Engine1b`. But most confusingly, the bearings `B1` and `B2` are switched in `Engine1b` compared to `Engine1a`. This is a tricky change to comprehend, since one of the bearings must break the kinematic-loop of the multi-body system; and the question is if turning them was required due to integration or numerical issues. As far as we can say that is not the case; `Engine1b` can be simulated with the bearings turned back without problems in Dymola. To make confusion complete, `Rod.r` and `Rod2/Rod1.r` are inversed between `Engine1a` and `Engine1bBase/Engine1b` (`r = {0, -0.2, 0}` vs. `{0, 0.2, 0}`) and must be turned to be consistent with the bearings switch. Although the sum of changes is correct, they are hard to comprehend. The incremental design of the engine is obscured behind a wall of model copying and modifications.



(a) `Engine1a` (existing MSL solution) **(b)** new `Engine1b` solution **(c)** new `Engine1b_analytic` solution

Figure 5. One cylinder engine scenario (proposed selective extension solution).

The alternative implementation of `Engine1b` and `Engine1b_analytic` based on selective extensions is much clearer. It is shown in Figure 5. Note the increase of diagram consistency; one can clearly see how starting from `Engine1a` the design is step-wise refined. The reason is that each design after the idealized start-model now inherits the diagram of the previous. Another advantage is that intentional modifications are evident. Consider for example the selective extension to implement `Engine1b`:

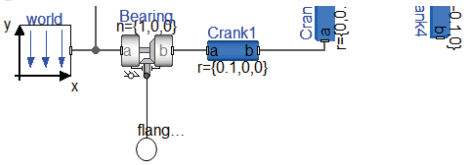
```
extends …MultiBody.Examples.Loops.Engine1a(
    Cylinder(useAxisFlange = true),
    Inertia(
      J = 0.1,
      phi(fixed = true, start = 0.001),
      w(fixed = true, start = 0)),
    cylPosition(r = {0.15,0.55,0}));
for each extends
  break connect(B2.frame_a, Piston.frame_b);
  break connect(B1.frame_b, Rod.frame_b);
  break connect(Rod.frame_a, B2.frame_b);
end for each extends;
// Add Rod3, Rod1 and gasForce and connect them…
```

The configuration differences to `Engine1a` can now be encapsulated in modifications as used to. Also the implementation of `Engine1b_analytic` is straight:

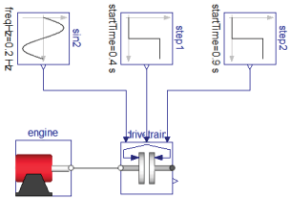
```
extends Engine1b;
for each extends
  break Cylinder;
  break B2; break B1; break Rod1; break Rod3;
end for each extends;
// Add the analytic solution and connect it…
```

The components comprised by the analytic solution are just replaced by it. Altogether, the new solution is much more consistent; changes like the unintended bearings switch and rod turning cannot just slip in.

As final challenge one could lift the engine to a component as shown in Section 4.1. Its fixed inertia, used for "startup", is problematic however. If used as component, an external inertia driven by the engine will be given instead. To that end, the inertia must be replaced by a flange-connector as shown in Figure 6 (a); the resulting engine component can be combined with the coupled clutch component of Section 4.1 to a simple powertrain as shown in Figure 6 (b).



(a) `Engine1b_analytic` component (excerpt)



(b) composition of engine component and clutches

Figure 6. Simple powertrain of engine and clutches.

5 Alternative designs

The proposed selective model extension is just a first step towards non-monotonic modelling in Modelica. Its final definition is open for discussion.

First of all, constraint (5) of Section 3.3.2 might be controversial; instead of a single `for each extends` block, several could be permitted. Deselections could be aligned with the `extends` clauses they deselect elements from. For example, the `extends` clause of Figure 2 (b) could look like (assuming the cut-off frequency of the filter has to be modified as well)

```
extends SMEE_Rectifier_Sampled(
  break voltageController,
  break setPointGain,
  filter(f_cut = 15));
```

Thus, all modifications and deselections regarding a base-model could be grouped with the respective

DOI 10.3384/ecp19157277

model extension. An advantage of aligning deselections with `extends` clauses is, that only elements of a specific base-model are excluded from inheritance. Of course, this gives rise to the question of consistency in case similarly named elements exist in several base-models; should the deselection of all be enforced or is it fine to deselect only a subset? The proposed semantic of `for each extends` always deselects all elements sharing a name, such that common base-model elements are consistently deselected; selection of a specific namesake requires its deliberate reintroduction, avoiding otherwise easy to miss indirect selections (indirect because the actually selected element is implicitly given by deselecting namesakes).

Another open issue is how fine-grained connections can be deselected. The definition of "matching" in Section 3.3.1 (c) is a very simple equivalence test just comparing the syntactic structure of the component references selecting the connected elements; the proposed semantic always deselects the complete matching connection. Since connectors can be hierarchical structured, including array elements, one could imagine more fine-grained deselections to rearrange parts of a structured base-class connection. Partial deselections of a structured connection could for example unlink only certain of its nested array and component elements. The graphical representation and editing of such deselections would be problematic however, since the structure of connections is not visible in Modelica's current diagram layer design.

Another limitation of the proposed solution is that only base-model elements can be deselected, but not their nested elements. Considering the crosscutting nature of Modelica Synchronous, qualified deselection might be very useful for synchronous adaptation. Like for structured connections however, again diagrammatic presentation and editing of nested component deselections would be problematic.

It is worthwhile to note that a relaxation of `replaceable`, by assuming all components are implicitly declared `replaceable` without type constraints, is insufficient for many cases handled by selective model extension. The problem is that `redeclare` cannot be used to consistently replace a *network* of components, as for example required to integrate the off-the-shelf induction machine controller of Figure 2 (b), where several original components must be removed, including their connections. To remove components in terms of re-declarations also is very cumbersome, not to speak of the consequences for the diagram layer which becomes cluttered with components representing actually removed and therefore not existing model parts that – quiet contrary – should not be shown at all.

Also the idea that all declarations and connections are implicitly conditional looks unsuitable; the parametric referencing for enabling and disabling would be tedious. The proposed selective model extension comprises this approach, just the other way around: instead of declaring everything conditional, it deselects by extension when actually required.

6 Conclusions

Engineering processes are typically not monotonic in terms that everything of an old design is taken when developing a new; some parts may be deliberately excluded and not present in the derived design. In terms of physics modeling in Modelica, such non-monotonic model variations are model-extensions with some original base-model features excluded from inheritance. Unfortunately, Modelica 3.4 is missing convenient means for structural non-monotonic modelling, which is a serious deficit the proposed selective model extension solves. Using selective extensions, no copying, changes or deliberate preparations on models are required to derive well-defined variants not preserving all of the original model structure. The presented concepts suffice to conveniently adapt models – including the examples of the Modelica Standard Library – for different synchronous application scenarios. And as shown in Section 4, selective model extension is also beneficial for a more natural engineering process with refinement-based model variation and adaptation. Artificial intermediate models, without physics simulation meaning, and system variation inconsistencies can be avoided; and non-monotonic interface adaptations required for cross-library integration incorporated. Particularly the latter will likely become an important future challenge, considering the likelihood of interface incompatibilities between libraries tends to increase with the success of the Modelica community and respective growing number of library suppliers. Another promising application area for selective model extension is model testing, particularly systematic fault introduction to simulate non-nominal behavior. The idea is to weave error sources into existing models, like noise-components intercepting a connection. Using selective extensions, the tested models do not have to be specifically prepared for fault injection; system parts can just be removed from inheritance and replaced by faulty – or even mock-up versions using external table-data – to inject misbehavior and configure the environment of error scenarios.

Acknowledgements

I am grateful for the help of Hans Olsson with the implementation of selective model extension in Dymola; his advices regarding the example scenario of Section 4 and proof reading have been very valuable. I also like to thank Hilding Elmqvist for his idea to add FMU extraction as valuable application case.

References

Modelica Association. *Modelica® - a unified object-oriented language for systems modeling: language specification version 3.4*, 2017.

Modelica Association. *Functional mock-up interface for model exchange and co-Simulation*, 2014.

Hilding Elmqvist, Martin Otter and Sven Erik Mattson. Fundamentals of synchronous control in Modelica. *Proceedings of the 9th International Modelica Conference*, 2012. doi:10.3384/ecp1207615.

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier and John Irwin. Aspect-oriented programming. *Lecture Notes in Computer Science*, 1241:220–242, 1997. doi:10.1007/BFb0053371.

Jørgen Lindskov Knudsen, Mats Löfgren, Ole Lehrmann Madsen and Boris Magnusson. *Object-oriented environments: the Mjølner approach*, Prentice Hall, 1993.

Martin Otter, Bernhard Thiele and Hilding Elmqvist. A Library for Synchronous control systems in Modelica. *Proceedings of the 9th International Modelica Conference*, 2012. doi:10.3384/ecp1207627.

Antero Taivalsaari. Classes vs. prototypes: some philosophical and historical observations. *Journal of Object-Oriented Programming*, 10(7):44–50, 1997.

Peri Tarr, Harold Ossher, William Harrison and Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. *Proceedings of the 21st International Conference on Software Engineering*, 1999. doi:10.1145/302405.302457.

Peter Wegner. Dimensions of object-based language design. *Proceedings of the 2nd conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1987. doi: 10.1145/38765.38823.

7 Appendix A

The following list is a rough estimation of examples of the Modelica Standard Library 3.2.2 that might be of interest for synchronous adaptation; about 60 existing continuous models have been identified. As criteria to consider a model relevant for synchronous adaptation, the containment of components modelling a controller has been chosen; further only *examples* – i.e., models extending `Modelica.Icons.Example` – have been considered. The potential examples of interest are (the actual example models are highlighted *green*):

```
Blocks
  Examples
    PID_Controller
    NoiseExamples
      ActuatorWithNoise
Electrical
  Machines
    Examples
      AsynchronousInductionMachines
        SwitchYD
        AIMC_Inverter
        AIMC_Conveyor
        AIMC_withLosses
      SynchronousInductionMachines
        SMR_Inverter
        SMPM_Inverter
        SMPM_CurrentSource
        SMPM_VoltageSource
        SMPM_Braking
        SMEE_Generator
        SMEE_LoadDump
        SMEE_Rectifier
  PowerConverters
    Examples
      ACDC
        Rectifier1Pulse
          Thyristor1Pulse_R
          Thyristor1Pulse_R_Characteristic
        RectifierBridge2Pulse
          HalfControlledBridge2Pulse
          ThyristorBridge2Pulse_R
          ThyristorBridge2Pulse_RL
          ThyristorBridge2Pulse_RLV
          ThyristorBridge2Pulse_RLV_Characteristic
          ThyristorBridge2Pulse_DC_Drive
        RectifierCenterTap2Pulse
          ThyristorCenterTap2Pulse_R
          ThyristorCenterTap2Pulse_RL
          ThyristorCenterTap2Pulse_RLV
          ThyristorCenterTap2Pulse_RLV_Characteristic
        RectifierCenterTapmPulse
          ThyristorCenterTapmPulse_R
          ThyristorCenterTapmPulse_RL
          ThyristorCenterTapmPulse_RLV
          ThyristorCenterTapmPulse_RLV_Characteristic
        RectifierBridge2mPulse
          HalfControlledBridge2mPulse
          ThyristorBridge2mPulse_R
          ThyristorBridge2mPulse_RL
          ThyristorBridge2mPulse_RLV
          ThyristorBridge2mPulse_RLV_Characteristic
          ThyristorBridge2mPulse_DC_Drive
        RectifierCenterTap2mPulse
          ThyristorCenterTap2mPulse_R
          ThyristorCenterTap2mPulse_RL
          ThyristorCenterTap2mPulse_RLV
          ThyristorCenterTap2mPulse_RLV_Characteristic
      DCAC
        SinglePhaseTwoLevel
          SinglePhaseTwoLevel_R
          SinglePhaseTwoLevel_RL
        MultiPhaseTwoLevel
          MultiPhaseTwoLevel_R
          MultiPhaseTwoLevel_RL
      DCDC
        ChopperStepDown
          ChopperStepDown_R
          ChopperStepDown_RL
        HBridge
          HBridge_R
          HBridge_RL
          HBridge_DC_Drive
Magnetic
  QuasiStatic
    FundamentalWave
      Examples
        BasicMachines
          InductionMachines
            IMC_Inverter
          SynchronousMachines
            SMPM_CurrentSource
            SMR_CurrentSource
Mechanics
  MultiBody
    Examples
      Systems
        RobotR3
          oneAxis
          fullRobot
Fluid
  Examples
    PumpingSystem
    DrumBoiler
      DrumBoiler
    ControlledTankSystem
      ControlledTanks
    AST_BatchPlant
      BatchPlant_StandardWater
    TraceSubstances
      RoomCO2WithControls
Thermal
  HeatTransfer
    Examples
      ControlledTemperature
```