Determining Over- and Under-Constrained Systems of Equations using Structural Constraint Delta

David Broman
Linköping University
davbr@ida.liu.se

Kaj Nyström
Linköping University
kajny@ida.liu.se

Peter Fritzson
Linköping University
petfr@ida.liu.se

Abstract

Computer aided modeling and simulation of complex physical systems, using components from multiple application domains, such as electrical, mechanical, and hydraulic, have in recent years witnessed a significant growth of interest. In the last decade, equation-based object-oriented (EOO) modeling languages, (e.g. Modelica, gPROMS, and VHDL-AMS) based on acausal modeling using Differential Algebraic Equations (DAEs), have appeared. With such languages, it is possible to model physical systems at a high level of abstraction by using reusable components.

A model in an EOO language needs to have the same number of equations as unknowns. A previously unsolved problem concerning this property is the efficient detection of over- or under-constrained models in the case of separately compiled models.

This paper describes a novel technique to determine over- and under-constrained systems of equations in models, based on a concept called structural constraint delta. In many cases it is also possible to locate the source of the constraint-problem. Our approach makes use of static type checking and consists of a type inference algorithm. We have implemented it for a subset of the Modelica language, and successfully validated it on several examples.

Categories and Subject Descriptors I.6.2 [*Simulation and Modeling*]: Simulation Languages

General Terms Algorithms, Languages

Keywords Equation-based, modeling, object-oriented, separate compilation, type checking, over-constrained, under-constrained

1. Introduction

Computer aided modeling and simulation have for years provided engineers in all disciplines with powerful tools to design and test complex systems in a faster and more cost-efficient way than physical prototyping. Computerized models also give the advantage of easy extraction of measurements from the model, even those that would be hard or even impossible to get from a physical system.

Historically, imperative implementation languages like Fortran and C have to some extent been replaced by specialized modeling platforms such as Simulink [8]. In recent years, new kinds of modeling languages have emerged, which combine the concept of object-orientation with specification of models using Differential Algebraic Equations (DAEs). We call these languages Equation-based Object-Oriented languages, or EOO languages for short. Modelica [6, 9] is an example of such a language. Other examples are gPROMS [10], χ [5], and VHDL-AMS [2].

While EOO languages provide attractive advantages, they also present new challenges in the areas of static analysis, type systems, and debugging. This paper deals with specific problems arising with EOO languages in two areas:

- Constraint checking of separately compiled components.
- Error detection and debugging.

1.1 Constraint Checking of Separately Compiled Components

A model in an EOO language is actually a system of equations describing the model's behavior. The existence of a single solution requires that the number of equations and variables (unknowns) are equal[1]. If the number of equations is greater than unknowns, the model is said to be *over-constrained*. Conversely, if the number of unknowns is greater than equations, it is *under-constrained*.

In an EOO model, variables and equations can be specified in different subcomponents of the model. To find out if a model has the same number of equations as variables, the model has traditionally been elaborated into a flat system of equations, where the number of variables and equations can be counted. However, this simple counting approach is not possible when one or more components in the model have been separately compiled.

Consider a simple model of a car, consisting of axis, gearbox, and an engine. In order to find out if the car model has the same number of equations as unknowns, we have to translate it into one large system of equations and count the number of variables and equations in that system. This is almost equivalent to a total recompilation of the entire car model and all its components. This in turn means that separate compilation of the subcomponents would have been completely unnecessary.

[1] This means that the incidence matrix associated with the system of equations is square, which is a necessary but not sufficient condition for the equation system to be structurally non-singular.

CORRECTED VERSION:
This author version has been updated with an error correction of the type inference algorithm. Changes has been made to Algorithm 2 on page 6 and in the last bullet item on page 7, left column.

1.2 Error Detection and Debugging

If a model intended for simulation has not the same number of equations as variables, it is an error. This can be detected after compiling the model into a system of equations. To locate and resolve the error, the system of equations must be inspected. Consider again the car model from Section 1.1. When the model is compiled (translated into equations), the user might be presented with an error message such as: “There are 20237 equations and 20235 variables”. Debugging the car model with only this message and a listing of equations and variables is extremely hard. There exist tools [4] and methods [1] that help the user in this process, but they require information of the model’s whole system of equations.

1.3 Contributions

The main contribution of this work is the novel concept of *structural constraint delta*, denoted C_Δ. Our approach makes use of static type checking and consists of a type inference algorithm, which determines if a model is under- or over-constrained without elaborating its subcomponents. This enables separate compilation of components in EOO languages. Furthermore, the concept also allows detection of constraint-errors at the subcomponent level and improves the possibilities of locating the source of the errors.

1.4 Outline

The remainder of this paper is structured as follows. Section 2 describes basic concepts and objectives of object-oriented equation-based modeling. Section 3 gives an overview of a Modelica compiler. Section 4 introduces a minimal EOO language called Featherweight Modelica (FM), its syntax and informal description of semantics and type system. Section 5 defines the concept of structural constraint delta, the algorithms used for constraint checking and debugging, and how these concepts fit into the FM language’s type system. Section 6 describes our prototype implementation, Section 7 discusses related work, and Section 8 presents conclusions of this paper.

2. Equation-Based Modeling in Modelica

In this section we illustrate several important concepts in modeling with EOO languages using the Modelica language as an example.

The basic structuring element in Modelica is the *class*. There are several restricted class categories with specific keywords, such as `model`, `record` (a class without equations), and `connector` (a record that can be used in connections). Just like in other OO languages, a class contains variables, i.e., class attributes representing data. These attributes are called *elements* of the class and can be instances of classes or built-in types. If the element is an instance of a `model`, this element is also called a `component`. The main difference compared with traditional OO languages is that instead of methods, Modelica primarily uses *equations* to specify behavior. Equations can be written explicitly, like `a=b`, or be inherited from other classes. Equations can also be specified by special `connect`-equations, also called *connections*. For example `connect(v1, v2)` expresses coupling between elements `v1` and `v2`. These elements are called *connectors* (also known as ports) and belong to the connected objects. This gives a flexible way of specifying the topology of a physical system.

2.1 Modelica Model of an Electric Circuit

As an introduction to Modelica, we present a model of an electrical circuit (Figure 1). A composite class like the `Circuit` model specifies the system topology, i.e., the components and the connections between the components. In the declaration of the resistor `R1`, `Resistor` is the class reference, `R1` is the component’s name, and `R=10` sets the default resistance, `R`, to `10`.

```
model Circuit
  Resistor  R1(R=10);
  Capacitor C(C=0.01);
  Resistor  R2(R=100);
  Inductor  L(L=0.1);
  VsourceAC AC;
  Ground    G;
equation
  connect(AC.p, R1.p);
  connect(R1.n, C.p);
  connect(C.n,  AC.n);
  connect(R1.p, R2.p);
  connect(R2.n, L.p);
  connect(L.n,  C.n);
  connect(AC.n, G.p);
end Circuit;
```

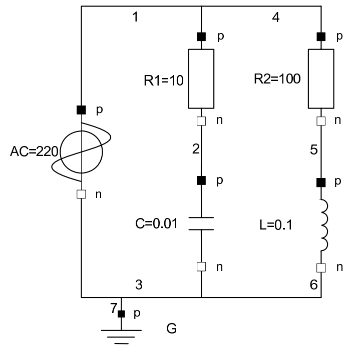

Figure 1. Modelica model of an electrical circuit.

2.2 Connector Classes

A connector must contain all quantities needed to describe an interaction. For electrical components we need the variables voltage `v` and current `i` to define interaction via a wire connection. A connector class is shown below:

```
connector Pin
  Real v;
  flow Real i;
end Pin;
```

A connect-equation `connect(R1.p,R2.p)` with `R1.p` and `R2.p` being instances of the connector class `Pin`, connects the two pins so that they form one node. This connect-equation generates two standard equality equations: `R1.p.v = R2.p.v` and `R1.p.i + R2.p.i = 0`. The first equation expresses that the voltage of the connected wire ends are the same. The second equation corresponds to Kirchhoff’s current law saying that the currents sum to zero at a node. The sum-to-zero equations are generated when the prefix `flow` is used. Similar laws apply to flow rates in a piping network and to forces and torques in mechanical systems. If a model contains an unconnected connector with a flow variable, the compiler will implicitly set this variable’s value to zero.

2.3 Base Classes and Inheritance

A common property of many electrical components is that they have two pins. Thus it is useful to define a “base” `TwoPin` component as follows:

```
model TwoPin "Superclass of components"
  Pin p, n;
  Voltage v;
  Current i;
equation
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end TwoPin;
```

This component has two pins p and n, a quantity v, that defines the voltage drop across the component and a quantity i that defines the current into the pin p, through the component and out from pin n. To define a model for an electrical capacitor we can now extend our base class TwoPin and add a declaration of a variable for the capacitance and the equation governing the capacitor's behaviour.

```
model Capacitor "Ideal electrical capacitor"
  extends TwoPin;
  Real C "Capacitance";
equation
  C*der(v) = i;
end Capacitor;
```

The keyword extends denotes inheritance from one or more base classes. Elements and equations are inherited from the parent.

2.4 Modification and Redeclaration

When extending or declaring an element, we can add *modification equations.* The simplest form of modification is assigning a value to a variable:

```
  Resistor R1(R=100);
```

It is also possible to alter the internal structure of a component when declaring or extending it, using redeclarations. The redeclare construct changes the class of the component being replaced. There are two restrictions on this operation:

1. The component we are replacing must be declared as replaceable.
2. The replacing class's type must be a subtype of the type of the component being replaced.

In this example, we create a model B from model A and at the same time change resistor R1 to be a TempResistor.

```
  model A
    replaceable Resistor R1(R=100);
  end model A;

  model B
    extends A(redeclare TempResistor R1);
  end B;
```

2.5 Acausal Modeling and Dynamic Systems

Modelica uses acausal modeling, which means modeling based on equations. Equations do not specify if a variable is used for input or output. In contrast, for assignment statements, variables on the left-hand side are always outputs (results) and variables on the right-hand side are always inputs. Thus, the causality of equations-based models is unspecified and becomes fixed only when the corresponding equation systems are solved. In practice, this means that when simulating a model, the user does not have to specify what variable he is interested in. The simulation will produce results for all variables present in the model.

Modelica is primarily used for modeling dynamic systems, where a model's behaviour evolves as a function of time. This means that all variables in a model have a value for every time step for which the model has been simulated. In addition, since we are working with DAEs, all derivatives of variables (denoted der(v) for the derivative of v) are derivatives with respect to time. An example of using the derivative function is shown in the Capacitor model in Section 2.3.

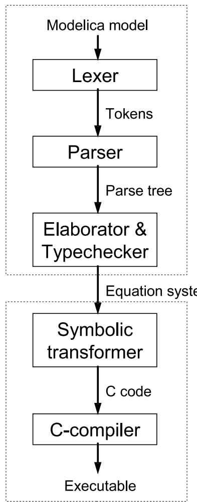

Figure 2. The structure of a Modelica compiler.

```
model B
  Real y;
  Real x;
equation
  y=der(x);
end B;


model C
  Real z=10;
  Real t;
equation
  t=z*2;
end C;


model A
  extends C(z=5);
  B b;
end A;
```

Figure 3. Example models to elaborate and type check.

3. The Modelica Compiler

In order to understand the current problem involved in separate compilation of Modelica models we must first explain how a typical Modelica compiler works. The structure of a typical compiler is depicted in Figure 2. It is common to view a Modelica compiler as consisting of two parts. The first part produces a system of equations and the second part produces an executable that solves this system of equations. The first steps in the compilation are scanning and parsing which transforms the Modelica source code into a parse tree. This parse tree is then *elaborated* into equations and variables.

3.1 Elaboration and Type Checking

First we need a few definitions; specific for this kind of language.

Definition 1 (Flat system of equations). *A flat system of equations is a set of declared variables of primitive types together with a set of equations referencing these variables.*

Definition 2 (Elaboration). *Elaboration is the task of producing a flat system of equations from the parse tree of a set of models.*

We will show how elaboration of a model is done by an example. The task is to elaborate model A in Figure 3. This means that from the code in model A, we should extract the corresponding system of equations. Examining model A, we find that it extends C. Our action is then to simply copy the contents of model C into our working copy of model A. The modification equation to variable z in the extends clause replaces the modification equation to variable z in C. All modifications are resolved as equations so the overriding modification z=5 is put in the equation section. The result so far is shown in model A1 in Figure 4.

We do not have to do anything about declarations of variables with primitive types. However, the component b must be elaborated since B is not of primitive type. We investigate model B and find that it contains the declarations Real y and Real x. These declarations and all equations

```
model A1
  Real z;
  Real t;
  B b;
equation
  z=5;
  t=z*2;
end A1;


model A2
  Real z;
  Real t;
  Real b.x;
  Real b.y;
equation
  b.y=der(b.x);
  z=5;
  t=z*2;
end A2;
```

Figure 4. Stepwise elaboration of model A from Figure 3.

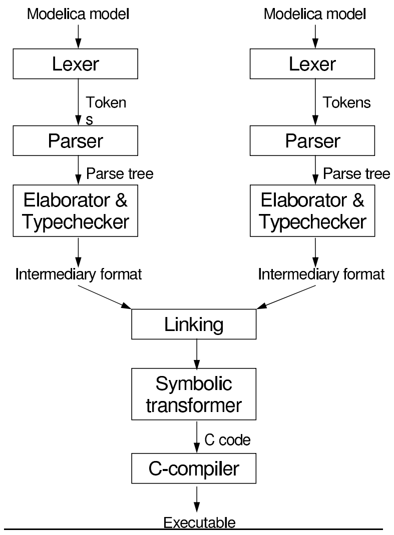

Figure 5. Separate compilation in Modelica.

in model B will now be inserted in our working model A with the prefix b. as we have now entered the namespace of the component b. The elaboration is now complete since there are only primitive types left in our working model. The final result of the elaboration is shown as model A2 in Figure 4.

In this compilation strategy, type checking is completely interleaved with the elaboration.

3.2 Symbolic Transformation and Code Generation

After elaboration, a number of operations are performed on the system of equations. Typically, the first step is to check that the number of equations and variables are equal. If this criterion is fulfilled, the compiler can go on to perform symbolic transformation tasks, such as BLT transformation [3]. We will not go into detail of these operations as they are not necessary for the understanding of this paper.

The symbolic transformation module then generates a program, usually in C code. The program uses a numerical solver such as DASSL [11] for solving the system of equations. The generated program can then be compiled with a C compiler which produces the executable which in turn will produce the simulation results.

3.3 Separate Compilation

Separate compilation in Modelica would ideally work as depicted in Figure 5.

The problem with separate compilation in Modelica is that while components may be separately compiled, it is hard to check if a model containing separately compiled components is under- or over-constrained. It seems that we must look at the entire elaborated model (flat system of equations) in order to determine this property.

We now return to the example of the car model mentioned in section 1.1. Let us assume that Engine, Gearbox, and Axis are very complex models consisting of more than 20000 equations developed by separate teams. It is clearly undesirable to recompile the entire system in order to check how it is constrained. Instead, we only want to elaborate the connect equations and to check the interfaces of the components.

3.4 Concluding Remarks

There are two deficiencies with the current practice in the Modelica compiler that we would like to stress.

1. Complete elaboration of all elements in a model is required to determine if the model is under- or over-constrained.
2. If the model turns out to be under- or over-constrained, it is very hard to find the bug since the error is detected at the level of flat system of equations rather than on a component/model level.

4. Featherweight Modelica

Modelica is a large and complex language that includes many concepts such as discrete simulation, algorithm sections, and functions, which are not central for our purpose. Consequently, we have designed and extracted a subset of the Modelica language, which models important aspects of the continuous and object-oriented parts of the language. We call this language Featherweight Modelica (FM). This section will informally present the language.

4.1 Syntax and Semantics

A model designed in FM can express continuous behavior by using Differential Algebraic Equations (DAEs). Reuse is achieved by the extends and redeclare constructs.

In Figure 6 the syntax grammar of FM is listed using a variant of extended Backus-Naur Form (EBNF). Alternatives are separated using the '|' symbol, optional arguments are given using square brackets ($[\cdots]$) and the curly brackets ($\{\cdots\}$) denote that the enclosed elements can be repeated zero or more times. Terminals are highlighted in bold-face.

The non-terminal *root* gives the starting point of a model definition. The metavariable M ranges over names of models and m over names of instances of models; C ranges over names of connectors and c over names of instances of connectors; R ranges over names of records and r over names of instances of records; x ranges over variable names of type Real. Note that subscribed numbers are used to differentiate between meta variables. All bold strings are keywords in the language except for Real, which is the built in type for $\mathbb{R}$.

The foundation of the language is the *class* concept, where model, connector, and record are special forms of classes. By observing the grammar, we can see that only models are allowed to have connections or contain elements that can be redeclared or modified. Connectors are the only classes whose instances can be part of a connect-equation, while Real types and record instances can be part of equations. Note that this can be seen in the grammar by considering the meta variables.

There are two kinds of prefixes: *access* and *modifiability.* Access prefixes state if an element in a model can be defined to be public or protected. The latter is only visible outside the model by a model extending from the class. The second prefix category is modifiability, defining how an element can be modified. Declaring an element replaceable makes it possible for a user to redeclare the element. Setting the prefix of an element to final means that the element can neither be modified nor redeclared. Only models can be redeclared and only Reals can be modified in FM.

```
          root  ::=  {model | connector | record}
         model  ::=  model M1
                     {extends M2 [modification] ;}
                     {[access] [modifiability]
                     (M3 m [modification] |
                     C c | R r | Real x [= lnum]) ;}
                     [equation {equation;}]
                     end M1 ;
     connector  ::=  connector C1 {extends C2 ;}
                     {[flow] Real x ;}
                     end C1 ;
        record  ::=  record R1 {extends R2 ;}
                     {(R3 r | Real x) ;}
                     end R1 ;
  modification  ::=  (modification' {, modification'})
 modification'  ::=  redeclare M m [modification]
                |    x = lnum
        access  ::=  public | protected
 modifiability  ::=  replaceable | modifiable
                |    final
      equation  ::=  connect(c1,c2) | e1 = e2
             e  ::=  e1 + e2 | e1 - e2 | e1 * e2 | e1 / e2
                |    -e | ( e ) | lnum | der(x) | x | r
                |    time | sin(e)
```

Figure 6. Syntax of Featherweight Modelica.

4.2 Type-Equivalence and Subtyping

Modelica is using a so called *structural type system* [12], where the type of a class is determined by the structure of its components. In contrast, other object-oriented languages, such as Java, are using primarily a *nominal type system*, where the name of the declared class identifies the type.

The Modelica language specification [9] is informally describing the semantics and type system of the language. From the specification, the following definition of *type equivalence* can be extracted:

Definition 3 (Type Equivalence). *Two types T and U are equivalent if T and U denote the same built-in type, or T and U are types of classes, T and U contain the same public declaration elements (according to their names), and the elements' types in T are equivalent to the corresponding element types in U.*

Note that a *class* `C` is not the same as the *type* of class `C`, since the type only represents the *interface* of the class and not the private *implementation* or *semantic* part, such as equations.

Besides type equivalence, the Modelica language defines subtyping relationships between types of classes.

Definition 4 (Subtyping). *For any types S and C, S is a supertype of C and C is a subtype of S if they are equivalent or if: every public declaration element of S also exists in C (according to their names) and those element types in S are supertypes of the corresponding element types in C.*

In the following text, we will use the notation of `C <: S`, meaning that the type of class `C` is a subtype of class `S`'s type.

```
model A              model B              model C
  Real p;              Real p;              extends A;
  Real c;              Real c;              Real q;
equation               Real q;            equation
  c = 2;             equation               q = p*p;
  der(p) = -c*p;       c = 2;             end C;
end A;                 der(p) = -c*p;
                     end B;
```

Figure 7. Three different Modelica models.

Now, consider the three models given in Figure 7. According to Definition 4, we can see that `B <: A` since the public elements `p` and `c` that exist in `A` also exist in `B`. We can see that `C` extends `A`, i.e., `C` inherits all components and equations from `A`. Furthermore, `C` defines an element `q`, which makes `C <: A`. In addition, since both `B` and `C` hold the same public elements, it can be concluded from Definition 3 that `B` and `C` are type equivalent.

Subtyping is a fundamental concept in many programming languages. Generally, it means that if a type S has all the properties of another type T, then S can be safely used in all contexts where type T is expected. This view of subtyping is often called *the principle of safe substitution* [12]. Now the question arise if this is true for the type system and examples described above. The main question is what we mean by *safe substitution* in the context of equation-based object-oriented languages. If we count the number of variables and equations in each of the models in Figure 7, we can see that model `A` has 2 variables and 2 equations, model `B` has 3 variables and 2 equations and finally model `C` has 3 variables and 3 equations. In the current type system of Modelica, both `B` and `C` are said to be safe replacements of `A`. However, in this case we know that replacing `A` with `C` gives us a potentially solvable system with 3 variables and 3 equations, but replacing `A` with `B` results in a under-constrained system with 3 variables and 2 equations, which will not give a unique solution. Can we after these observation still regard `B` as a safe replacement of `A`? We think not, and will in the next subsections propose a solution.

5. The Approach of Structural Constraint Delta

In this section, we will present an approach that addresses the problem of determining under- and over-constrained components without performing elaboration. We start by giving a definition:

Definition 5 (Structural Constraint Delta, C_Δ). *Given an arbitrary class C, containing components, equations, and connections, the type of C has a defined integer attribute called structural constraint delta, C_Δ. This attribute states, for C and all its subcomponents, the integer difference between the total number of defined equations and variables.*

The term *structural* indicates that the equations and variables are counted as they are declared in the model. For example, two linearly dependent equations in an equation system will still be counted as two separate equations. Hence, $C_\Delta = 0$ for a system of equations does not guarantee a unique solution, it will only indicate that a single solution might exist. If $C_\Delta < 0$, we have an under-constrained problem with more unknowns than equations, which might give an infinite number of solutions. If $C_\Delta > 0$, we have an over-constrained system of equations, which most likely will not give a unique solution. However, since the algorithm for computing $C_\triangle$ does not check if equations are linearly independent or not, a system with $C_\triangle > 0$ may be solvable. To be able to guarantee that a system of equations has a unique solution, complete knowledge of the entire system of equation must be available. Since this is obviously not possible when inspecting components separately, the value of $C_\triangle$ only provides a good indication whether a system of equations has a unique solution or not.

For example, if $C_\triangle$ is to be calculated for the types of the models given in Figure 7, the difference between the number of equations and variables in the model gives the value of $C_\triangle$. In this case, $C_\triangle = 0$ for `A` and `C`, but $C_\triangle = -1$ for `B`. Since our models so far only contain variables and equations, calculating $C_\triangle$ is straightforward. However, if a model contains hundreds of subcomponents, using connections, connectors, and records, the resulting flattened system might consist of thousands of equations. To be able to formulate algorithms for calculating $C_\triangle$, we need another definition:

Definition 6 (Constraint Delta Effect, $E_\triangle$). *Let C be an arbitrary class containing two elements c1 and c2 that are instances of classes C1 and C2, which contain only elements and no equations or connections. Given an equation or connection E located in C representing a relation between c1 and c2, the constraint delta effect $E_\triangle$ is a type attribute of both C1 and C2, which states the effect E has when computing $C_\triangle$ of C.*

Note that $C_\triangle$ is not the same as $E_\triangle$. Simply stated, we say that $E_\triangle$ of two elements represents the change of the current model's $C_\triangle$ when an equation or connection is introduced between the two elements. For example, if we in model `B` in Figure 7 introduce a new equation `q = 2 * p`, this equation will have the effect of changing model `B`'s $C_\triangle$ from -1 to 0. Therefore, involved variables `q` and `p`, are said to have $E_\triangle = 1$ (or to be precise; the attributes to the types of the elements). However, we will soon see that elements do not always have $E_\triangle = 1$.

5.1 Algorithms for Computing $C_\triangle$ and $E_\triangle$

In this section, we present algorithms for calculating $C_\triangle$ and $E_\triangle$. Even if the algorithms for calculating the type attributes $C_\triangle$ and $E_\triangle$ could be stated by using a formal type system, we have chosen to illustrate the algorithm more informally using pseudo-code algorithms. The main reasons for this are that the Modelica language itself has currently no formal semantics or type system and the target audience of this paper is not only computer scientists, but also engineers from the modeling and simulation community.

It is important to stress that $C_\triangle$ and $E_\triangle$ are defined as attributes to the *types* of the classes, and not for the classes themselves. This implies that when calculating the value for a specific class `C`, we do not need to recursively calculate $C_\triangle$ and $E_\triangle$ for each subelement, since they are already defined by the type of the elements. The process of calculating $C_\triangle$ and $E_\triangle$ is a form of *type inference*, i.e., the type attributes are inferred from equations given in the class and types of the elements in the class.

The algorithm for computing $C_\triangle$ is given in Algorithm 1. This algorithm uses a help function defined in Algorithm 2. The algorithm for computing $E_\triangle$ is listed in Algorithm 3. Note that the indentation of the algorithms is significant and delimits blocks for the `foreach`, `if`, and `switch` statements.

To make the algorithms more easy to follow, the following help functions are defined:

Algorithm 1: Compute $C_\triangle$ of a class

```
Input: An arbitrary $Class$
Output: $C_\triangle$ of the class
1  $C_\triangle \leftarrow 0$
2  switch $Class$ do
3    case model
4      foreach $e \in$ getElements($Class$) do
5        $C_\triangle \leftarrow C_\triangle +$ getDelta($e$)
6        if hasDefaultValue($e$) then
7          $C_\triangle \leftarrow C_\triangle + 1$
8        foreach $m \in$ getModifiedElements($e$) do
9          if not hasDefaultValue($m$) then
10           $C_\triangle \leftarrow C_\triangle + 1$
11     foreach $e \in$ getEquations($Class$) do
12       $C_\triangle \leftarrow C_\triangle +$ getEffect($e$)
13     foreach $c \in$ getConnectors($Class$) do
14       $P_{outside} \leftarrow$ FALSE
15       $P_{inherited} \leftarrow$ FALSE
16       if not isVisited($c$) then
17         traverseConnectorGraph($c$)
18         if $P_{outside}$ then
19           $C_\triangle \leftarrow C_\triangle +$ getOutsideAdjustment($c$)
20     foreach $b \in$ getBaseClasses($Class$) do
21       foreach $m \in$ getModifiedElements($b$) do
22         if not hasDefaultValue($m$) then
23           $C_\triangle \leftarrow C_\triangle + 1$
24       $C_\triangle \leftarrow C_\triangle +$getDelta($b$)
25   case record
26     foreach $e \in$ getElements($Class$) do
27       $C_\triangle \leftarrow C_\triangle +$ getDelta($e$)
28     foreach $b \in$ getBaseClasses($Class$) do
29       $C_\triangle \leftarrow C_\triangle +$ getDelta($b$)
30   case connector
31     foreach $e \in$ getElements($Class$) do
32       if not hasFlowPrefix($e$) then
33         $C_\triangle \leftarrow C_\triangle +$ getDelta($e$)
34       foreach $b \in$ getBaseClasses($Class$) do
35         $C_\triangle \leftarrow C_\triangle +$ getDelta($b$)
36   case variable
37     $C_\triangle \leftarrow -1$
38 end
```

Algorithm 2: traverseConnectorGraph(c_1)

```
Input: Connector $c_1$ from which graph traversal starts
Output: Global variables $P_{outside}$, $P_{inherited}$, and $C_\triangle$
1  if ((isOutside($c_1$) and isInherited($c_1$)) or ((isOutside($c_1$)
2      or isInherited($c_1$)) and ($P_{outside}$ or $P_{inherited}$))
     then typeCheckingFailed()
3  else
4    markAsVisited($c_1$)
5    $P_{outside} \leftarrow P_{outside}$ or isOutside($c_1$)
6    $P_{inherited} \leftarrow P_{inherited}$ or isInherited($c_1$)
7    foreach $c_2 \in$ getAdjacencyConnectors($c_1$) do
8      if not isVisited($c_2$) then
9        $C_\triangle \leftarrow C_\triangle +$ getEffect(getTypeOf($c_2$))
10       traverseConnectorGraph($c_2$)
```

Algorithm 3: Compute $E_\triangle$ of a class

```
Input: An arbitrary $Class$
Output: $E_\triangle$ of the class
1  $E_\triangle \leftarrow 0$
2  switch $Class$ do
3    case record
4      foreach $e \in$ getElements($Class$) do
5        $E_\triangle \leftarrow E_\triangle +$ getEffect($e$)
6      foreach $b \in$ getBaseClasses($Class$) do
7        $E_\triangle \leftarrow E_\triangle +$ getEffect($b$)
8    case connector
9      foreach $e \in$ getElements($Class$) do
10       if hasFlowPrefix($e$) then
11         $E_\triangle \leftarrow E_\triangle -$getEffect($e$)
12       else  $E_\triangle \leftarrow E_\triangle +$ getEffect($e$)
13     foreach $b \in$ getBaseClasses($Class$) do
14       $E_\triangle \leftarrow E_\triangle +$ getEffect($b$)
15   case variable
16     $E_\triangle \leftarrow 1$
17 end
```

- **getAdjacencyConnectors** (c) - the set of connectors that are directly connected to c by connect-equations declared in the local class.
- **getBaseClasses** (C) - the set of types for the base classes to C.
- **getConnectors**(C) - the set of accessible connectors that are used by connections in class C. All connectors are initially marked as unvisited.
- **getDelta**(t) - attribute $C_\triangle$ part of type t.
- **getElements**(C) - the set of types for elements part of class C.
- **getEquations**(C) - the set of equations part of the local class C, excluding connect-equations and equations from base classes. Each element in the set represents the type of the expressions declared equal by the equation.
- **getEffect**(t) - the attribute $E_\triangle$ part of type t.
- **getModifiedElements**(e) - the set of elements' types in e, which is modified by modification equations.
- **getOutsideAdjustment**(c) - an integer value representing adjustments to be made if connector c is part of a connector set that is connected to an outside connector. The integer value is equal to the positive number of flow variables inside connector c.
- **getTypeOf**(c) - the type of connector c.
- **hasDefualtValue**(e) - TRUE if element type e has a defined default value.
- **hasFlowPrefix**(e) - TRUE if element e is prefixed with keyword `flow`.
- **isInherited**(c) - TRUE if connector c is inherited from a base class.
- **isVisited**(c) - TRUE if connector c is marked as visited.
- **isOutside**(c) - TRUE if connector c is seen as an outside connector in the local class.
- **markAsVisited**(c) - mark connector c as visited.
- **typeCheckingFailed**() - terminates the type checking, since two outside or inherited connectors are connected, or a connected connector is both outside and inherited.

5.1.1 Computing $C_\triangle$ - Equations, Inheritance, and Modification

We start by illustrating the algorithms using trivial examples, where the models only contain equations, records, and variables. Consider the following FM listing:

```
record R        C△=-2 E△=2   model B          C△=0
  Real p;       C△=-1 E△=1     Real y=10;C△=0E△=1
  Real q;       C△=-1 E△=1   end B;
end R;
                             model M          C△=-1
model A         C△=-3          extends A(p=1);C△=-2
  R r1;         C△=-2 E△=2    B b1(y=20);     C△=0
  R r2;         C△=-2 E△=2    B b2;           C△=0
  Real p;       C△=-1 E△=1    equation
equation                        b1.y = p;
  r1 = r2;                   end M;
end A;
```

Model `M` extends from model `A`, which implies that all equations and elements in `A` will be merged into `M`. Model `A` contains two instances of record `R`. If each of these models were to be compiled separately, we would need to calculate $C_\triangle$ for each of the models without any knowledge of the internal semantics of the subcomponents, i.e., the equations. Calculated $C_\triangle$ and $E_\triangle$ for every class and element are given to the right in the listing.

Consider Algorithm 1, which takes an arbitrary class as input and calculates the $C_\triangle$ value for this class. First, we can see that calculating $C_\triangle$ of a record simply adds the $C_\triangle$ value for each element (rows 26-27), which in the case of record `R` gives $C_\triangle = -2$ since `R` holds 2 variables. In Algorithm 3, we can see that calculating the effect of `R` gives $E_\triangle = 2$. But what does this mean? Recall that $E_\triangle$, given in Definition 6, states the effect on $C_\triangle$ when connecting two elements. In model `A`, an equation `r1 = r2` is given, which uses record `R`. This equation will after elaboration generate two equations, namely `r1.p = r2.p` and `r1.q = r2.q`, which is why $E_\triangle$ for `R` is 2. The rest of the procedure for computing $C_\triangle$ of model `A` should be pretty straightforward by following Algorithm 1. Note that only $C_\triangle$ and not $E_\triangle$ is given for models, since models are not allowed to be interconnected.

The more interesting aspects of calculating $C_\triangle$ in this example are shown in model `M`. First of all, we can see that model `M` extends from `A`, which results in that $C_\triangle$ of `A` is added to $C_\triangle$ of `M` (see rows 20-24 in Algorithm 1). Since variable `p` is modified with `p=1`, we see that $C_\triangle$ is increased by $E_\triangle$ of the type of `p`, i.e., `Real`. Hence, the $C_\triangle$ contribution from base class `A` is -2. The $C_\triangle$ value for model `B` is 0. When instantiated to element `b1` in model `M`, its element `y` is modified with `y=20`. However, this modification does not effect $C_\triangle$, since `y` already has a default value (see rows 8-10 in Algorithm 1). Finally, we can see that the total calculation of `M` will result in a $C_\triangle$ value of -1.

5.1.2 Computing $C_\triangle$ - Connectors, Connections, and Flow Variables

Consider the source code listing and graphical representation given in Figure 8. Model `M` contains components `a` and `b`, which are instances of model `K`. Each model consist of several connector instance, all instances of a connector class `C`.

The semantics of the Modelica language distinguish between *outside connectors* and *inside connectors*, where the former are connector instances denoting the border of a model, e.g., `oc1` and `oc2`, and the latter represents connectors available in local components, e.g., `a.ic1`, `a.ic2`, `b.ic1`, and `b.ic2`. Note that a connector instance can be seen as both an outside and an inside connector, depending which model is being processed. In this example we are looking at model `M`.

Calculating C_Δ of connector `C` can be achieved by using rows 30-35 in Algorithm 1. On row 32, we can see that C_Δ is only added if the variable has not got a flow prefix. The reason for this is that an unconnected flow variable has always a defined default equation, setting its value to 0. Hence, introducing a flow variable gives one extra variable and one equation, i.e., $C_\Delta = 0$. Further inspection of the algorithm, yields $C_\Delta = -2$ for model `K`.

Calculating C_Δ of `M` is more complicated. On row 13 in Algorithm 1 it is stated that we iterate over all involved connectors, in this case `a.ic1`, `a.ic2`, `b.ic1`, `b.ic2`, `oc1`, and `oc2`. Variable $P_{outside}$ becomes TRUE if the algorithm has passed an outside connector, and $P_{inherited}$ becomes TRUE if it has passed an inherited element. The latter case will not be illustrated in this example. The first thing to notice is that the connector graph is traversed by using the recursive function traverseConnectorGraph(), listed in Algorithm 2. The algorithm performs a *depth-first search* visiting each connector (vertex) only once, by marking it as visited. Note that function traverseConnectorGraph() has side effects and updates the variables $P_{outside}$, $P_{inherited}$, and C_Δ. Each connect-equation (edge) in the graph contribute to the C_Δ of the class being computed, by adding E_Δ of a connector in the connection (see row 9 in Algorithm 2). Since all connectors traversed in one iteration of the foreach loop are connected (row 13-19 in Algorithm 1), all types of the connectors hold the same value of E_Δ.

By using Algorithm 3, rows 9-12, we can see that $E_\Delta = 0$ for connector `C`. Consequently, all the connections in model `M` will not change the value of C_Δ. Why is this the case? We know that connecting non-flow variables will always result in an extra equation, i.e., for non-flow variables, E_Δ must be 1. However, when connecting two flow variables, one equation is added, but two default equations are removed. For example in `connect(a.ic2, b.ic1);`, the two default equations `a.ic2.x=0` and `b.ic1.x=0` are removed and replaced with the sum-to-zero equation `a.ic2.x + b.ic1.x = 0`. Hence, the effect of connecting two flow variables is $E_\Delta = -1$.

There are several aspects covered by the algorithms, which we will not be able to explain in detail in this paper, due to space limitations. The following items briefly describe some of these issues:

- If cycles appear in the connector graph, there exists a redundant connect-equation which does not contribute to the value of C_Δ. For example, if connections `connect(oc1,b.ic1)` and `connect(a.ic1,a.ic2)` would be introduced in `M`, one connection would be redundant. This issue is handled by making sure that connectors are only visited once (see rows 7-10 in Algorithm 2.)
- Connecting an inside connector to an outside connector does not give the same effect on C_Δ as connecting inside to inside. For example, when connecting `oc1` to a local connector inside `M`, the default variable `oc1.x=0` will not be removed. This default equation will only be removed when `oc1` is connected outside model `M`, i.e., when another model is using `M` as a component. This issue is managed on rows 18-19 in Algorithm 1.
- The algorithm does not allow direct or indirect connections between outside connectors. For example a connection `connect(oc1,b.ic2)` would generate a type checking error (see row 1-2 in Algorithm 2). The same semantics hold for connections between connectors inherited from base classes. We use this conservative approach since without it, the type of a class must be extended with information regarding the connectors that are connected.

```
model K        connector C
  C ic1;         flow Real x;
  C ic2;         Real y;
end K;         end C;

model M
  K a;
  K b;
  C oc1;
  C oc2;
equation
  connect(a.ic1, oc1);
  connect(a.ic2, b.ic1);
  connect(b.ic2, oc2);
end M;
```

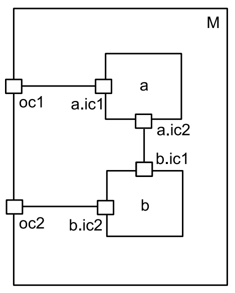

Figure 8. Model `M` with inside connectors (e.g. `a.ic1` and `b.ic2`) and outside connectors (`oc1` and `oc2`).

5.2 Extending the Type System with C_Δ

The latter sections described how we can calculate C_Δ and E_Δ of classes, resulting in value attributes for types in the language. However, this is of no use if we do not apply this new information to the type system. A new extended version of the Featherweight Modelica language, denoted FM$_\Delta$, is defined by extending Definition 3 and Definition 4 for type-equivalence and subtyping with the following definitions:

Definition 7 (Type-equivalence and C_Δ). *For any types T and U to be type-equivalent, Definition 3 must hold and the C_Δ-value of T and U must be equal.*

Definition 8 (Subtyping and C_Δ). *For any types S and C, S is a supertype of C and C is a subtype of S if Definition 4 holds and the C_Δ-value of S is equal to that of C.*

Hence, the extended language FM$_\Delta$ guarantees that the difference between declared variables and equations does not change when using the rule of subsumption. If we recall the models listed in Figure 7, we can now see that model `C` is a subtype of model `A`, but model `B` is not.

6. Prototype Implementation

To validate and verify our algorithms, a prototype Featherweight Modelica Compiler (`fmc`) was implemented consisting of a type-checker for FM$_\Delta$, where C_Δ and E_Δ automatically are inferred and represented as attribute to the types. The prototype compiler was implemented as a batch-program, which takes a FM$_\Delta$ `.mo`-file (containing FM$_\Delta$ models) as input and returning to standard output the pretty-printed type of the last model defined in the `.mo`-file.

To validate the correctness of our solution, the following procedure has been used:

1. Create relevant models in FM$_\Delta$.
2. Run the prototype compiler for FM$_\Delta$ on the models. The output is the listed type of the model including C_Δ information.

3. Elaborate the model and manually inspect the flat Modelica code generated by the compilers Dymola version 6 [4] and OpenModelica version 1.4.1 [7].

We will now analyze, by using a simple circuit example, how the concept of structural constraint delta attacks the problems of constraint checking with separately compiled components, and error detection and debugging. In the examples, fmc and Dymola version 6 are used when testing the models.

6.1 Constraint Checking of Separately Compiled Components

Consider the following listing, stating the model Resistor, a connector Pin and a base class TwoPin:

```
model TwoPin              connector Pin
  Pin p;                    Real v;
  Pin n;                    flow Real i;
  Real v;                 end Pin;
  Real i;
equation                  model Resistor
  v = p.v - n.v;            extends TwoPin;
  0 = p.i + n.i;            Real R = 100;
  i = p.i;                equation
end TwoPin;                 R*i = v;
                          end Resistor;
```

When using fmc, each of these models are separately type checked. For example, when typechecking model Resistor, model TwoPin and connector Pin are not elaborated. Instead, only the types of TwoPin and Pin are used. This information is available after these classes are compiled.

Below the output generated by fmc is listed, with some pretty printing added for readability:

```
model classtype C_△=0
  public final connector objtype C_△=-1 E_△=0
    nonflow Real objtype v;
    flow Real objtype i;
  end p;
  public final connector objtype C_△=-1 E_△=0
    nonflow Real objtype v;
    flow Real objtype i;
  end n;
  public modifiable Real objtype v;
  public modifiable Real objtype i;
  public modifiable Real objtype* R;
end
```

The lines above represent the type of model Resistor. Note the difference made between *class type* (the type of a class that can be instantiated), and a *objtype* (the type of an object that has been instantiated by a class). The type's of elements p and n have $C_\triangle = -1$ and $E_\triangle = 0$. The latter indicates that when the Resistor model is used by connecting p or n, $C_\triangle$ will not change. Finally, we can see that that $C_\triangle = 0$ for the whole type of Resistor.

Now, if the following code is added to our .mo-file, we have a complete model named Circuit that we can simulate.

```
model Ground           model VsourceAC
  Pin p;                 extends TwoPin;
equation                 Real VA = 220;
  p.v = 0;               Real f = 50;
end Ground;              Real PI = 3.1416;
                       equation
                         v = VA*sin(2*PI*f*time);
                       end VsourceAC;
```

```
model Inductor     model Circuit
  Pin p;           protected
  Pin n;             replaceable Resistor R1(R=10);
  Real v;            replaceable Inductor L(L=0.1);
  Real i;            VsourceAC AC;
  Real L = 1;        Ground G;
equation           equation
  L*der(i) = v;      connect(AC.p, R1.p);
end Inductor;        connect(R1.n, L.p);
                     connect(L.n, AC.n);
                     connect(AC.n, G.p);
                   end Circuit;
```

Trying to simulate the above model Circuit in the commercial Modelica environment Dymola, the error feedback states that it is not possible to simulate it because there are 22 equations and 25 variables in the flattened equation system.

Executing the model in fmc, we get the response that model circuit has $C_\triangle = -3$, which corresponds to the message Dymola reported. Note that Dymola had to elaborate all the models to a flattened system of equation to get to this result. fmc on the other hand could use the separately type checked components and just use the types of these components to get the same result. Hence, this example illustrates how our approach can be used to enable separate compilation of components.

6.2 Error Detection and Debugging

Now the following question arise: How can we know where the problem is located? The user needs to either analyse the model code or to inspect the flat system of equations. In both cases, this problem seems hard to manage.

If we run this model in fmc, we get the following type information for model Circuit (for readability, parts of the type are replaced by a dotted line):

```
model classtype C_△=-3
  protected replaceable model objtype C_△=0
    ...
  end R1;
  protected replaceable model objtype C_△=-3
    ...
  end L;
  protected modifiable model objtype C_△=0
    ...
  end AC;
  protected modifiable model objtype C_△=0
    ...
  end G;
end
```

Analyzing the type information, it indicates that it is component L, which is an instance of Inductor that probably causes the under-constrained system. After a closer look, we notice that Inductor is not extending from TwoPin, as it should. After replacing the old Inductor model with

```
model Inductor
  extends TwoPin;
  Real L = 1;
equation
  L*der(i) = v;
end Inductor;
```

it is possible to simulate the model.

Now, let us assume that we want to build a larger model having model Circuit as a subcomponent. However, this time we do not want to use a Resistor in Circuit. Instead, the goal is to redeclare R with a temperature dependent resistor called TempResistor. Consider the following models:

```
model TempResistor
  extends TwoPin;
  Real R;        // Resistance at. reference temp.
  Real RT=0;     // Temp. dependent resistance
  Real Tref=20;  // Reference temperature
  Real Temp;     // Actual temperature
equation
  v = i * (R + RT * (Temp-Tref));
end TempResistor;

model Circuit2
  extends Circuit(redeclare TempResistor R1(R=35));
end Circuit2;
```

Trying to simulate this model in Dymola results in a flattened model with 28 variables and 27 equations, which cannot be simulated. By elaborating all components and analyzing the system of equations, Dymola hints that R1 is structurally singular.

However, using fmc, this model does not even pass the type checker. The compiler reports that C_Δ for the original type is 0 (Resistor), but the redeclaring model's type is -1 (TempResistor). Hence, the subtyping rule is not legal and the redeclaration is incorrect. The following listing shows a correct redeclaration, where the temperature parameter Temp has been assigned a value.

```
model Circuit3
  extends Circuit
    (redeclare TempResistor R1(R=35, Temp=20));
end Circuit3;
```

Consequently, our approach finds the incorrect model at an early stage during type checking. Furthermore, since the type checking was performed on precompiled models, there is no need for elaborating the model's subcomponents. Hence, this approach is not only useful for separate compilation, but also for users when locating errors in models.

7. Related Work

We have used the Modelica language as an example to explain the problems associated with over- and under-constrained systems. These problems arise in languages using hierarchical modeling with components, where the component semantics contain DAEs. While it is trivial to count equations in a simple model, we have seen that the complexity increases when introducing connect semantics, existing in e.g. the χ [5] language. Both flow variables, used in e.g. VHDL-AMS [2] (called through) and inheritance part of e.g. gPROMS [10], complicate matters further.

The Modelica language includes all these concepts, and there exist methods for locating errors at the level of flat system of equations [1]. The Modelica tool Dymola [4] detects constraint-errors at the flat system of equations, and can sometimes also pinpoint the errors. One downside with these approaches is that the entire model must be elaborated, making separate compilation difficult.

An attractive simplification related to the C_Δ concept would be to require all separately compiled models to have the same number of equations as unknowns, i.e., $C_\Delta = 0$. However, it is an open question if this approach is not too conservative for expressing models in the general case.

To the best of our knowledge, no solution has previously been presented for any applicable language that determines if a model is under- or over-constrained, without elaborating the model.

8. Conclusions

We have presented the concept of structural constraint delta (C_Δ) for equation-based object-oriented modeling languages. Algorithms for computing C_Δ were given, and it was shown how C_Δ is used to determine if a model is under- or over-constrained *without* having to elaborate a model's components. We have also illustrated how the concept of C_Δ allows the user to detect and pinpoint some model errors. The concept has been implemented for a subset of the Modelica language and successfully tested on several models.

Acknowledgments

Thanks to Peter Bunus, John Wilander, Thomas Schön, and Åsa Broman for many useful comments. This research work was funded by CUGS (the National Graduate School in Computer Science), SSF in the Visimod-II project, MathCore Engineering, and by Vinnova in the NETPROG Safe & Secure Modeling and the GridModelica projects.

References

[1] Peter Bunus and Peter Fritzson. Automated Static Analysis of Equation-Based Components. *SIMULATION*, 80(7–8):321–245, 2004.

[2] Ernst Christen and Kenneth Bakalar. VHDL-AMS - A Hardware Description Language for Analog and Mixed-Signal Applications. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 46(10):1263–1272, 1999.

[3] Iain. S. Duff and John K. Reid. Algorithm 529: Permutations To Block Triangular Form [F1]. *ACM Trans. Math. Softw.*, 4(2):189–192, 1978.

[4] Dynasim. Dymola - Dynamic Modeling Laboratory with Modelica (Dynasim AB). http://www.dynasim.se/ [Last accessed: 8 May 2006].

[5] Georgina Fábián. *A Language and Simulator for Hybrid Systems.* PhD thesis, Technische Universiteit Eindhoven, the Netherlands, 1999.

[6] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1.* Wiley-IEEE Press, New York, USA, 2004.

[7] Peter Fritzson, Peter Aronsson, Håkan Lundvall, Kaj Nyström, Adrian Pop, Levon Saldamli, and David Broman. The OpenModelica Modeling, Simulation, and Development Environment. In *Proceedings of the 46th Conference on Simulation and Modeling (SIMS'05)*, pages 83–90, Trondheim, Norway, 2005.

[8] MathWorks. The Mathworks - Simulink - Simulation and Model-Based Design. http://www.mathworks.com/products/simulink/ [Last accessed: 15 May 2006].

[9] Modelica Association. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling - Language Specification Version 2.2*, February 2005. Available from: http://www.modelica.org [Last accessed: 29 March 2006].

[10] M. Oh and Costas C. Pantelides. A modelling and Simulation Language for Combined Lumped and Distributed Parameter Systems. *Computers and Chemical Engineering*, 20(6–7):611–633, 1996.

[11] Linda R. Petzold. A Description of DASSL: A Differential/Algebraic System Solver. In *IMACS Trans. on Scientific Comp., 10th IMACS World Congress on Systems Simulation and Scientific Comp.*, Montreal, Canada, 1982.

[12] Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, 2002.