Type-Based Structural Analysis for Modular Systems of Equations

Henrik Nilsson, University of Nottingham, UK, *nhn@cs.nott.ac.uk*

SNE Simulation Notes Europe SNE 19(1), 2009, 17-28, doi: 10.11128/sne.19.tn.09923

This paper investigates a novel approach to a type system for modular systems of equations; i.e., equation systems constructed by composition of individual equation system fragments. The purpose of the type system is to ensure, to the extent possible, that the composed system is solvable. The central idea is to attribute a *structural type* to equation system fragments that reflects which variables occur in which equations. In many instances, this allows over- and underdetermined system fragments to be identified separately, without first having to assemble all fragments into a complete system of equations. The setting of the paper is equation-based, non-causal modelling, specifically Functional Hybrid Modelling (FHM). However, the central ideas are not tied to FHM, but should be applicable to equation-based modelling languages in general, like Modelica, as well as to applications featuring modular systems of equations outside the field of modelling and simulation.

Introduction

An important question in the context of equation-based modelling is whether or not the system of equations describing the modelled entity is solvable. In general, this can only be answered by studying the complete system of equations, and often not even then, except by attempting to solve the equations through simulation.

This is problematic. Models are usually modular, i.e. described by combining small systems of equations into larger ones. Being able to detect problems with individual parts or their combinations without first having to put together a complete system model is generally desirable. Moreover, a system may be *structurally dynamic*, meaning that the system of equations describing its behaviour *changes* over time. This implies that the question of the solvability cannot be addressed prior to simulation.

However, establishing that a system of equations *definitely is not solvable* can be almost as helpful. Fortunately there are criteria necessary (but not sufficient) for solvability that can be checked more easily and that are applicable to model fragments. A simple example is that the number of variables (unknowns) and equations must agree. For example, Modelica as of version 3.0 [12] enforces this constraint for model fragments (and thus for a model as a whole) so as to enable early detection of common modelling mistakes. Keeping track of the variable and equation balance is also the idea behind the structural constraint delta type system [2] with similar aims.

This paper is a preliminary investigation into an improved type-based (and thus compile-time) analysis for determining when (fragments of) systems of equations *cannot* be solved. The goal is to provide improved precision compared with just counting variables and equations by attributing a *structural type* to systems of equations reflecting which variables occur in which equations. A type-based approach is adopted as that is a natural way of ensuring that model fragments can be checked in isolation. This is particularly important for structurally dynamic systems where parts of the system change over time. However, as long as the *types* of the parts remain unchanged, and are reasonably informative, a meaningful analysis can still be carried out statically, at compile-time.

The development is carried out in the context of Functional Hybrid Modelling (FHM) [14, 15], as this provides a small and manageable modelling language framework that helps keeping the focus on the essence of the problem. FHM itself is still in an early stage of development. However, the central ideas put forward in this paper are not tied to FHM, but should be applicable to equation-based modelling languages like Modelica in general, as well as to applications featuring modular systems of equations outside the field of modelling and simulation. In effect, FHM is mainly used as a convenient and concise notation for modular systems of equations.

The rest of the paper is organised as follows. Section 1 provides general background and discusses related work. Section 2 provides an overview of FHM in the interest of making this paper relatively self-contained. Section 3 then develops the idea of structural types for modular systems of equations. As an example, this is applied to a simple electrical circuit in Section 4. Finally, Section 5 discusses future work and Section 6 gives conclusions.

1 Background and related work

Object-oriented modelling languages like Modelica [12] allow models to be developed in a *modular* fashion: systems of equations describing individual components are composed into larger systems of equations describing aggregates of components, and ultimately into a complete model of the system under consideration. As with software in general, such modularity is key to addressing the complexity of large-scale development as it allows large problems to be broken down into smaller ones that can be addressed independently, enables reuse, etc.

Of course, it is possible that mistakes are made during the development of a model. If so, it is desirable to catch such mistakes early. In a modular setting, this means checking whether a component in isolation is inherently faulty, and whether two or more components are being composed appropriately. As a result, mistakes can be localised effectively, meaning it becomes a lot easier to find and correct them. In contrast, mistakes that only become evident once a system has been fully assembled are usually a lot harder to pinpoint as the symptom in itself often is not enough to suggest any particular part of the system as the root of the problem. Even more problematic is a situation where problems only reveal themselves in use, as this means the system is unreliable.

A good way to catch errors early is to employ the notion of *types*. An entity has some particular type if it satisfies the properties implied by that type. A *type system* then governs under which conditions typed entities may be combined, and determines what properties the combined entity satisfies, i.e. its type. As a simple example, consider the type `Integer`. If an entity has type `Integer`, this means that this entity satisfies the property of being an integer. Moreover, a rule of the type system would establish that *any* two entities satisfying the property of being integers can be combined using arithmetic addition into a new entity that also is an integer. This example is trivial, but as we will see, it is possible to capture much more complex properties through suitably defined types.

An important aspect of a type system is that it works solely on the basis of the *types* of the combined entities, without referring to any specific entity *instances*. This makes it possible to establish various properties of a combined entity before knowing exactly what all its parts are. This in turn allows for all manner of useful parametrisations, systems with dynamically evolving structure, etc.

This paper is concerned with equation systems properties for establishing whether a system can be solved or not. One necessary but not sufficient condition for solvability is the variable and equation balance: globally, the number of variables to solve for and the number of equations must be equal. Languages like Modelica naturally enforce this. Since version 3.0 [12], Modelica has adopted the even stricter criterion that (in essence) variables and equations must be *locally* balanced, i.e. balanced on a per component basis. Thus, in a sense, the property of being balanced is implicitly part of the type of a component in Modelica 3.0, as all well-typed components are balanced. Naturally, if all components of a model are locally balanced, this implies that the model is globally balanced.

Of course, a locally imbalanced model might still be globally balanced. To allow such models (without deferring *all* checking until a model has been fully assembled), it is necessary to *explicitly* make the variable and equation imbalance part of the type of a component. This was suggested by Nilsson *et al.* [14] and, independently, by Broman *et. al.* [2], who developed the idea in detail by integrating the notion of a "structural constraint delta" into the types of components.

Unfortunately, ensuring that the number of variables and equations agree only gives relatively weak assurances. As a simple example, consider the following system of equations, where f, g, and h are known functions, and x, y, and z are variables:

$$\begin{aligned} f(x,y,z) &= 0 \\ g(z) &= 0 \\ h(z) &= 0 \end{aligned}$$

The number of equations and variables agree. Yet it is clear that we cannot hope to solve this system of equations: x and y occur only in one equation, but we need two equations to have a chance to determine both of them. Moreover, z occurs alone in two of the equations, meaning that it may be impossible to find a value of z that satisfies them both. What we have in this case is an *underdetermined* system of equations for x and y (one equation, two variables), and an *overdetermined* system of equations for z (two equations, one variable). Note that it was possible to establish the unsolvability of this system by just considering its *structure*: which variables occur in which equations. This can be formalised through the notion of a *structurally singular* system of equations:

Definition 1 (Structually singular system of eqs.)
A system of equations is *structurally singular* iff it is not possible to put the variables and equations in a one-to one correspondence such that each variable occurs in the equation it is related to.

We now simply observe that a system of equations that is structurally singular is unsolvable.

Languages like Modelica ensure that models are not structurally singular as simulation is not possible if this is the case. However, in Modelica, this check is not carried out on a per component basis, but only once the system has been fully expanded into a “flat” system of equations. To the best of this author’s knowledge, this is also the case for all similar languages. As a result, if it turns out that the final model is structurally singular, it can be very difficult to find out what the origin of the problem is.

To help overcome this difficulty, Bunus and Fritzson proposed a method to help localising the cause of any structural singularity [3, 4]. Their idea is to view the system of equations as a bipartite graph where the variables constitute one set of nodes, the equations the other set of nodes, and there is an edge between a variable and an equation if the former occurs in the latter. See Figure 1(a) and 1(b). They then use the Dulmage and Mendelsohn canonical decomposition algorithm [6] to partition the flat system of equations into three parts: one overdetermined, one underdetermined, and one where the variables and equations match up. This information is then used to help diagnose the problem and suggest remedies.

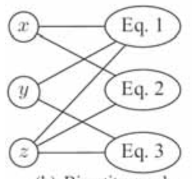
Still, it would be an advantage if mistakes that *inevitably* are going to lead to structural singularities can be flagged up early, without first having to fully expand a model. This is true in particular for structurally dynamic systems: since the system of equations describing the behaviour of the system change over time, there is no one fully expanded system in this case. This is the kind of systems we ultimately hope to address in the context of our work on Functional Hybrid Modelling [14, 15].

This paper investigates an approach to early detection of structural singularities. The basic idea is to attribute types to components such that these types characterise the *structure* of the underlying system of equations used to represent a component, or more precicely, the structure of the equations that constitute its *interface*. We refer to this as the *structural type* of the component. The fundamental idea is similar to the structural constraint delta approach suggested by Broman *et al.*. However, the structural type is much richer: instead of a single number reflecting the variable and equation imbalance, the structural type details which variables occur in which equations. That is, the structural type is essentially a bipartite graph as in the work by Bunus and Fritzson, or it can be viewed as an *incidence matrix*: see Figure 1(c).We will freely switch between these two points of view in the following.

It turns out, though, that it often will be necessary to approximate the information on which variables occur in which equations. Thus the approach of this paper is not a complete alternative to error diagnosis on the final, flat system of equations as suggested by Bunus and Fritzson, but rather complementary to it.

$$\begin{aligned} f(x,y,z) &= 0 \quad (1)\\ g(x,z) &= 0 \quad (2)\\ h(y,z) &= 0 \quad (3)\end{aligned}$$

(a) System of equations



(b) Bipartite graph

$$\begin{array}{cc} & \begin{array}{ccc} x & y & z \end{array} \\ \begin{array}{c} \text{Eq. 1} \\ \text{Eq. 2} \\ \text{Eq. 3} \end{array} & \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} \end{array}$$

(c) Incidence matrix

Figure 1. A system of equations and its corresponding structural representations.

2 Functional hybrid modelling

Functional Hybrid Modelling (FHM) [14, 15] is a generalisation of the central ideas of *Functional Reactive Programming* (FRP) [18]. In FRP, a functional programming language is extended with constructs for reactive programming and *causal*, hybrid, modelling, specifically *signals* (time-varying values) and functions on signals. This has proved to yield a very flexible and expressive framework for many different kinds of reactive and modelling applications [13, 9, 5, 8]. The FHM approach is similar, but *relations on signals* are added to address *non-causal* modelling.

The salient features of FRP and FHM relevant for this paper are covered in the rest of this section. The ideas are illustrated with a simple circuit example. This example is also used later in this paper. Note that FHM is currently being developed: no complete implementation exists yet. However, as explained earlier, it provides a convenient setting for this work.

2.1 Fundamental concepts

FRP is a conceptual framework. A number of concrete implementations exists. Here, we will briefly consider Yampa [13], which is most closely related to FHM. Yampa is based on two central concepts: signals and signal functions. A signal is a function from time to a value; conceptually:

$$\mathit{Signal}\ \alpha \approx \mathit{Time} \rightarrow \alpha$$

(The conceptual nature of this definition is indicated by $\approx$. $\rightarrow$ is the infix type constructor for function types.) Time is continuous, and is represented as a non-negative real number. The type parameter α

specifies the type of values carried by the signal. For example, the type of a varying electrical voltage might be *Signal Voltage*.

A *signal function* is a function from *Signal* to *Signal*:

$$SF\ \alpha\ \beta\ \approx\ Signal\ \alpha\ \rightarrow\ Signal\ \beta$$

When a value of type *SF* α β is applied to an input signal of type *Signal* α, it produces an output signal of type *Signal* β. Signal functions are *first class entities* in Yampa. Signals, however, are not: they only exist indirectly through the notion of signal function. Additionally, signal functions satisfies a causality requirement: at any point in time, the output must not depend on future input (this is *temporal* causality, a notion distinct from the notion of causality in "non-causal modelling.")

The output of a signal function at time t is uniquely determined by the input signal on the interval $[0,t]$. If a signal function is such that the output at time t only depends on the input at the very same time instant t, it is called *stateless*. Otherwise it is *stateful*.

2.2 First-class signal relations

A natural mathematical description of a continuous signal function is that of an ODE in explicit form. A function is just a special case of the more general concept of a *relation*. While functions usually are given a causal interpretation, relations are inherently non-causal. Differential Algebraic Equations (DAEs), which are at the heart of non-causal modelling, express dependences among signals without imposing a causality on the signals in the relation. Thus it is natural to view the meaning of a DAE as a non-causal *signal relation*, just as the meaning of an ODE in explicit form can be seen as a causal signal function. Since signal functions and signal relations are closely connected, this view offers a clean way of integrating non-causal modelling into an Yampa-like setting.

Similarly to the signal function type SF of Yampa (Section 2.1), the type *SR* α stands for a relation on a signal of type α. Like signal functions, signal relations are first class entities, as will become clear in the following. Specific relations use a more refined type; e.g., for the derivative relation **der** we have the typing:

der :: *SR* (*Real*, *Real*)

Since a signal carrying pairs is isomorphic to a pair of signals, we can understand **der** as a binary relation on two real-valued signals. Signal relations are constructed as follows:

sigrel *pattern* **where** *equations*

The pattern introduces *signal variables* that at each point in time are bound to the *instantaneous* value of the corresponding signal. Given a pattern *p* of type *t*, *p* :: *t*, we have:

sigrel *p* **where** … :: *SR t*

Consequently, the equations express relationships between instantaneous signal values. This resembles the standard notation for differential equations in mathematics. For example, consider $x' = f(y)$, which means that the instantaneous value of the derivative of (the signal) x at every time instant is equal to the value obtained by applying the function f to the instantaneous value of y.

There are two styles of basic equations:

$e_1 = e_2$
$sr \lozenge e_3$

where e_i are expressions (possibly introducing new signal variables), and *sr* is an *expression* denoting a signal relation. We require equations to be well-typed. Given $e_i :: t_i$, this is the case iff $t_1 = t_2$ and $sr :: t_3$.

The first kind of equation requires the values of the two expressions to be equal at all points in time. For example:

$f\ x\ =\ g\ y$

where *f* and *g* are ordinary, pure functions (we follow standard functional programming practice and denote ordinary function application simply by juxtapositioning, without any parentheses.)

The second kind allows an arbitrary relation to be used to enforce a relationship between signals. The symbol $\lozenge$ can be thought of as *relation application*; the result is a constraint which must hold at all times. The first kind of equation is just a special case of the second in that it can be seen as the application of the identity relation. Thus, with I denoting the identity relation, an equation $e_1 = e_2$ could also be written I $\lozenge$ (e_1, e_2). For another example, consider a differential equation like $x' = f(x)$. Using the notation above, this equation can be written:

der $\lozenge$ $(x, f\,x\,y)$

where **der** is the relation relating a signal to its derivative. For notational convenience, we will often use a notation closer to standard mathematical practice:

der $x = f\,x\,y$

The meaning is exactly as in the first version. Thus, in the second form, **der** is *not* a pure function operat-

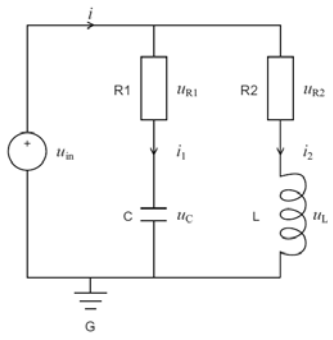


Figure 2. A simple electrical circuit.

ing only on instantaneous signal values. It is a (stateful) signal function operating on the underlying signal.

We illustrate the ideas above by modelling the electrical circuit in Figure 2 (adapted from [11]). The type *Pin* is a record type describing an electrical connection. It has fields *v* for voltage and *i* for current.

```
twoPin :: SR (Pin, Pin, Voltage)
twoPin = sigrel (p, n, u) where
    u = p.v − n.v
    p.i + n.i = 0
resistor :: Resistance → SR (Pin, Pin)
resistor r = sigrel (p, n) where
    twoPin ◊ (p, n, u)
    r * p.i = u
inductor :: Inductance → SR (Pin, Pin)
inductor l = sigrel (p, n) where
    twoPin ◊ (p, n, u)
    l * der p.i = u
capacitor :: Capacitance → (Pin, Pin)
capacitor c = sigrel (p, n) where
    twoPin ◊ (p, n, u)
    c * der u = p.i
```

The resistor, inductor and capacitor models are defined as extensions of the *twoPin* model. This is accomplished using functional abstraction rather than any Modelica-like class concept. Note how parameterized models are defined through functions *returning* relations, e.g. *resistor*. Since the parameters (like *r* of *resistor*) are normal function arguments, *not* signal variables, their values remain unchanged throughout the lifetime of the returned relations (in Modelica terms, they are parameter variables). As signal relations are first class entities, signal relations can be parameterized on other signal relations in the same way.

To assemble these components into the full model, a Modelica-inspired **connect**-notation is used as a convenient abbreviation for connection equations. In FHM, this is just syntactic sugar that is expanded to basic equations: equality constraints for connected potential quantities and a sum-to-zero equation for connected flow quantities. In the following, connect is only applied to *Pin* records, where the voltage field is declared as a potential quantity whereas the current field is declared as a flow quantity.

We assume that a voltage source model *vSourceAC* and a ground model *ground* are available in addition to the component models defined above. Moreover, we are only interested in the total current through the circuit, and, as there are no inputs, the model thus becomes a *unary* relation:

```
simpleCircuit :: SR Current
simpleCircuit = sigrel i where
    resistor 1000 ◊ (r1p, r1n)
    resistor 2200 ◊ (r2p, r2n)
    capacitor 0.00047 ◊ (cp, cn)
    inductor 0.01 ◊ (lp, ln)
    vSourceAC 12 ◊ (acp, acn)
    ground ◊ gp
connect acp r1p r2p
connect r1n cp
connect r2n lp
connect acn cn ln gp
i = r1p.i + r2p.i
```

There is no need to declare variables like *r1p*, *r1n*: their types are inferred. Note the signal relation expressions like *resistor* 1000 to the left of the signal relation application operators ◊.

As an illustration of signal relation application, let us expand resistor 1000 ◊ (*r1p*, *r1n*) using the definitions of twoPin and *resistor*. The result is is the following three equations, where u1 is a fresh variable:

```
u1 = r1p.n − r1n.v
r1p.i + r1n.i = 0
1000 * r1p.i = u1
```

2.3 Dynamic structure

Yampa can express highly structurally dynamic systems. Ultimately, we hope to integrate as much of that functionality as possible into FHM. As a basic example, switching among two different sets of equations as a Boolean signal changes value might be expressed as follows:

```
switch b
when False
    equations₁
when True
    equations₂
```

If the type system approach outlined in this paper is to work for FHM, we need to consider how to handle such constructs from a type perspective. This is done in Section 3.4. There are many other outstanding problems related to implementation of structurally dynamic systems. But those are outside the scope if this paper.

3 Structural types for signal relations

We now define the notion of structural type and show how it enables structural analysis to be carried out in a modular way, without having to first expand out signal relations to "flat" systems of equations. The key difficulty is abstraction of structural types, and consequently the section mostly focuses on that aspect.

3.1 The structural type

In essence, a signal relation is an *encapsulated* system of equations. When a signal function is applied, these equations impose constraints on signals in scope at the point of application through the variables of the signal relation *interface*. A larger system of equations is thus formed, composed from equations contributed by each applied signal relation.

Let us consider a simple example:

$$\begin{array}{l} foo :: SR\,(Real, Real, Real) \\ foo = \textbf{sigrel}\ (x_1, x_2, x_3)\ \textbf{where} \\ \quad f_1\, x_1\, x_2\, x_3 = 0 \\ \quad f_2\, x_2\, x_3 \quad = 0 \end{array}$$

Let us assume a context with five variables, u, v, w, x, y, and let us apply *foo* twice in that context:

$$\begin{array}{l} foo \lozenge (u, v, w) \\ foo \lozenge (w, u + x\ , v + y) \end{array}$$

The result, obtained by substituting the variables u, v, w, x, y into the equations of *foo*, is the following system of equations:

$$\begin{array}{ll} f_1\, u\, v\, w & = 0 \\ f_2\, v\, w & = 0 \\ f_1\, w\, (u + x\ )\, (v + y) & = 0 \\ f_2\, (u + x\ )\, (v + y) & = 0 \end{array}$$

Note that each application of *foo* contributed two equations to the composed system, each for a subset of the variables to the right of the relation application operator ◊.

As discussed in Section 1, the aim is now to analyse the structure (which variables occur in which equations) of the composed system in order to identify situations that definitely will result in over- or under-determined systems of equations.

However, for a variety of reasons, it is not desirable to assume that this can be done by simply unfolding the applied relations as was done above. In the context of FHM, what goes to the left of ◊ is a signal relation *expression* that may involve parameters that are not known at compile time, thus preventing the expression from being evaluated statically. Or the exact contribution of the applied signal relation might not be known for other reasons, for example due to separate compilation or because it is structurally dynamic.

Thus, we are only going to assume that the *type* of the applied signal relation is known. To enable structural analysis, the type of signal relations is enriched by a component reflecting its structure.We refer to this as the *structural type* of the signal relation.

Definition 2 (Structural type of system of equations)
The structural type of a system of equations is the incidence matrix of that system. It has one row for each equation, and one column for each variable in scope—only "unknown" signal variables are of interest here, not parameters or "known" (input) signal variables. An occurrence of a variable in an equation is indicated by 1, a non-occurrence by 0.

Note that Definition 2 concerns systems of equations. For a *signal relation*, i.e. an *encapsulated* system of equations, the structural type is limited to the equations *contributed* by the signal relation and the variables of its interface. If the interface includes records of signal variables, like *Pin* of the simple circuit example in Section 2.2, then each field counts as an independent variable. We defer a precise definition until section 3.3.

As an example, consider the signal relation *foo* above. Its type, including the structural part, is:

$$foo :: SR(Real, Real, Real)\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

3.2 Composition of structural types

Now let us consider composition of structural types. The overall structural type for a sequence of equations is obtained by simply joining the incidence matrices for the individual equations as the same set of variables is in scope across all equations.

The structural type for a basic equation of the form

$$e_1 = e_2$$

is a single-row matrix indicating which variables occurs in expressions e_1 and e_2.

The structural type for the second form of equation, signal relation application, is more interesting. The general form of this kind of equation is:

$sr \lozenge (e_1, e_2, \ldots, e_i)$

where e_1, e_2, …, e_i are expressions over the signal variables that are in scope. These expressions and their relation to the variables in scope can also be represented by an incidence matrix, with one row for each expression and one column for each variable. The incidence matrix of the signal relation *application* is then obtained by Boolean matrix multiplication—which is understood as Boolean conjunction, $\wedge$ (logical "and"), and addition as Boolean disjunction, $\vee$ (logical "or")—of the structural type of the applied signal relation and the incidence matrix of the right-hand side expressions.

Returning to the example from the previous section, the incidence matrix of the right-hand side of the application

$foo \lozenge (u, v, w)$

in a context with five signal variables u, v, w, x , y is

$$\begin{array}{ccccc} u & v & w & x & y \\ \hline 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{array}$$

(where the columns have been labelled for clarity). Multiplying the structural type of foo with this matrix yields:

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix} \begin{array}{c} \begin{array}{ccccc} u & v & w & x & y \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} \end{array} = \begin{array}{c} \begin{array}{ccccc} u & v & w & x & y \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

Similarly, for $foo \lozenge (w, u + x , v + y)$, we obtain:

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix} \begin{array}{c} \begin{array}{ccccc} u & v & w & x & y \end{array} \\ \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix} \end{array} = \begin{array}{c} \begin{array}{ccccc} u & v & w & x & y \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \end{pmatrix} \end{array}$$

The complete incidence matrix for the two applications of *foo* is thus

$$\begin{array}{c} \begin{array}{ccccc} u & v & w & x & y \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \end{pmatrix} \end{array}$$

Compare with the fully expanded system of equations in the previous section.

3.3 Abstraction over structural types

In the previous section, we saw how to obtain the overall structural type of a composition of signal relations given the structural types of the involved signal relations. The next step is to consider how to encapsulate a system of equations in a signal relation. It is often the case that the set of variables in the interface of a signal relation, the *interface variables* is a proper subset of the variables that are in scope. A signal relation may thus abstract over a number of *local variables*. This, in turn, means that a number of the equations at hand *must* be used to solve for the local variables: the local variables are not going to be in scope outside the signal relation, and thus it is not possible to add further equations for them later.

The available equations are thus going to be partitioned into *local equations*, those that are used to solve for local variables, and *interface equations*, those that are contributed to the "outside" when the signal relation is applied. This immediately presents an opportunity to detect instances of over- and under-determined systems of equations for the local variables on a per signal relation basis. However, it also presents a very hard problem as the partitioning is not uniquely determined, which in general implies that a signal relation does not have unique best structural type.

To illustrate, consider encapsulating the example from the previous section in a signal relation where only the variables u and y appear in the interface:

bar = **sigrel** (u, y) **where**
  $foo \lozenge (u, v, w)$
  $foo \lozenge (w, u + x , v + y)$

Recall the incidence matrix of the encapsulated system:

$$\begin{array}{c} \begin{array}{ccccc} u & v & w & x & y \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \end{pmatrix} \end{array}$$

Three of the underlying equations are needed to solve for the local variables v, w, and x , the remaining one is the interface equation. But the only equation that *cannot* be chosen as the interface equation is number 2, as no interface variable occurs in this equation. Projecting out the columns for the interface variables for the the incidence matrices for the three possible choices of interface equation yields

$$\begin{array}{cc} u & y \\ (1 & 0) \end{array} \quad \begin{array}{cc} u & y \\ (1 & 1) \end{array} \quad \begin{array}{cc} u & y \\ (1 & 1) \end{array}$$

The last two possibilities are equivalent, so this leaves us with two possible structural types: the signal relation *bar* can either provide a single equation in which the first variable of the interface occurs, or it can provide an equation in which both interface variables occurs, depending on the chosen equation partitioning in *bar*.

A modelling language compiler will decide on a specific partitioning. But this choice is typically dictated by intricate numerical considerations and often also by the usage context. As it is essential that type checking is compositional, it is clear that the partitioning must be done independently of usage context. And to ensure that the type system is independent of arbitrary implementation choices, as well as reasonably easy to understand for the end user, it is clear that the partitioning should not depend on low-level numerical considerations either.

There are two approaches for dealing with the situation. One is to *accept* that a signal relation can have more than one structural type. This paper does not explore that avenue as there is a risk that it would lead to a combinatorial explosion of possibilities to consider. Still, it should not be ruled out. The other approach is to decide on a suitable notion of "best" structural type. Then, if a signal relation has more than one possible structural type, choose the best one, if this is a uniquely determined choice, otherwise approximate all best types with a type that is better than them all, but still as informative as possible, and take this approximation as the structural type of the signal relation.

We are going to adopt a notion of "best" that reflects the observation that an equation is more useful the more variables that occur in it (as this gives more flexibility when choosing which equation to use to solve for which variable). We are further going to assume that an implementation is free to make such a best choice. The latter might not be the case, but we should then keep in mind that the objective of the type system is *not* to guarantee that a system of equations *can* be solved, but to detect cases where a system of equations definitely *cannot* be solved. Assuming a freedom of choice is thus a safe approximation.

Definition 3 (Subsumed variable)
Let V_1 and V_2 be sets of variables. V_1 is *subsumed* by V_2 iff $V_1 \setminus V_2 = \varnothing$.

Definition 4 (Subsumed structural type)
Let s_1 and s_2 be structural types. s_1 is *subsumed* by s_2 iff there exists a permutation of the rows of the incidence matrix for s_2 such that the variables of each row of the incidence matrix for s_1 are subsumed by the variables of the corresponding row of the permuted incidence matrix for s_2. The subsumed relation on structural types is denoted by the infix symbol $\leq$.

Definition 5 (Best structural types)
Let S be a set of structural types. The *best structural types* in S is the set

$$\{s \mid s \in S \wedge \neg(\exists s' \in S : s \leq s)\}$$

Returning to the signal relation *bar* above, we find that it actually has a single best structural type since

$$\begin{array}{cc} u & y \\ (1 & 0) \end{array} \leq \begin{array}{cc} u & y \\ (1 & 1) \end{array}$$

The complete type of *bar* is thus:

$$bar :: SR(Real, Real)\begin{pmatrix}1 & 1\end{pmatrix}$$

As an example of a case where there is not any best type, consider

$$s_1 = \begin{pmatrix}1 & 0 & 1 \\ 1 & 0 & 0\end{pmatrix}, \quad s_2 = \begin{pmatrix}0 & 0 & 1 \\ 1 & 1 & 1\end{pmatrix}$$

Note that $s_1 \not\leq s_2$ and $s_2 \not\leq s_1$. Neither is better than the other, and the best structural types of $S = \{s_1, s_2\}$ is S.

What is needed if there is more than one best type is to find an approximation in the form of an upper bound that subsumes them all. Clearly such a bound exists: just take the incidence matrix with all 1s, for example. That corresponds to an assumption that each equation can be used to solve for any variable, meaning that we are back to the approach of counting equations and variables. However, to avoid loosing precision unnecessarily, a *smallest upper bound* should be chosen. As the following example shows, there may be more than one such bound, in which case one is chosen arbitrarily.

Consider the two structural types:

$$\begin{pmatrix}1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1\end{pmatrix}, \quad \begin{pmatrix}0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1\end{pmatrix}$$

Upper bounds can be constructed by taking the union of the first incidence matrix and all possible row permutations of the second one. As there are only two rows, we get two upper bounds:

$$\begin{pmatrix}1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1\end{pmatrix}, \quad \begin{pmatrix}1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1\end{pmatrix}$$

Neither is smaller than the other. However, they are both as small as possible, as removing a single 1 from any matrix means it will not subsume one or the other of the original matrices. Thus, in general, the least upper bound of structural types under the subsumed ordering is not uniquely determined.

We can now give a definition of the structural type of a signal relation:

Definition 6 (Structural type of a signal relation)
The *structural type of a signal relation* with a body of m equations over n variables, of which i variables occur in the interface, if that type exists, is an $(m-(n-i))\times i$ incidence matrix that is a least upper bound of the structural types of all possible choices of interface equations.

The following algorithm determines the structural type of a signal relation when one exists, or reports an error otherwise. We claim this without proof, leaving that as future work:

Arguments

1. Structural type s for the system of equations of the body of the signal relation in the form of an $m\times n$ incidence matrix (m equations, n variables).
2. The set V of variables, $|V| = n$, and a mapping from variables to the corresponding column number of the incidence matrix.
3. The set I of interface variables of the signal relation.

Result

- If successful, an $(m-(n-|I|))\times|I|$ incidence matrix representing the structural type of the signal relation.
- Otherwise, an indication of the problem(s): under- or overdetermined system of local equations; overdetermined system of interface equations.

Algorithm

1. Let $L = V\setminus I$ be the set of local variables. Partition s into three parts:
   - s_L: rows corresponding to equations over variables in L only, the *a priori local equations*;
   - s_I: rows corresponding to equations over variables in I only the a priori interface equations;
   - s_M: remaining rows, corresponding to equations over mixed interface and local variables.

   Let m_L, m_I, m_M be the number of rows of s_L, s_I, and s_M respectively. (Note that the a priori local equations can *only* be used to solve for local variables, whereas the a priori interface equations can *only* be used to solve for interface variables.)
2. Let $k = |L| - m_L$. k is the number of equations in addition to local ones that are needed to solve for all local variables.
   a. If $k < 0$, report "overdetermined local system of equations".
   b. If $k > m_M$, report "underdetermined local system of equations".
3. Initialise S_I to $\varnothing$
4. Choose k rows from s_M in all possible ways $\binom{m_M}{k}$ possibilities, $m_M \geq k$). For each such choice:
   a. Partition s_M into $s_{L'}$ containing the k chosen rows and $s_{I'}$ containing the remaining rows.
   b. Consider s_L and $s_{L'}$ restricted to the local variables L as a bipartite graph and compute a maximum matching using the standard augmenting path algorithm [1, pp. 246–250]. Check if the size of the matching is equal to $|L|$. If yes, this means that each variable in L can be paired with a row from s_L or $s_{L'}$ in which it occurs, which is a necessary condition for using the equations corresponding to the rows from s_L or $s_{L'}$ to solve for the local variables.
   c. Consider s_I and $s_{I'}$ restricted to the interface variables I as a bipartite graph and compute a maximum matching using the standard augmenting path algorithm. Check if the size of the matching is equal to the number of rows of s_I and $s_{I'}$, i.e. $m_I + m_M - k$. If yes, then this means that all equations corresponding to the rows of s_I and $s_{I'}$ can be used simultaneously to solve for one of the interface variables. This is a necessary condition for ensuring that the interface equations contributed by the signal relation does not constitute an overdetermined system.
   d. If both checks above passed, then this particular choice of k rows is *valid*.
   e. For each valid choice, add $s_{I'}$ restricted to the variables I to $S_{I'}$.
5. If $S_{I'} = \varnothing$, it is not possible to solve for the local variables and/or the interface equations contributed by the signal relation are going to be overdetermined. Report the problem.

6. Determine the best structural types $S_{I''}$ of $S_{I'}$.
7. Let $s_{I''}$ be a least upper bound of $S_{I''}$.
8. The incidence matrix obtained by joining s_I and $s_{I''}$ is the structural type of the signal relation, i.e. a least upper bound of the structural types of all possible choices of interface equations.

3.4 Structurally dynamic systems

To conclude the development, we briefly consider how to handle structurally dynamic systems, for example of the type illustrated in section 3.3. Clearly, the structural types of the equations in the different branches could be different. However, at any point in time, the choice of which equations that are active is determined by the condition of the **switch**-construct. Thus, the structural type of the entire **switch**-construct is the *greatest lower bound* of the structural types of the branches, as that is the only thing which is guaranteed at all points in time. One may also want to impose additional consistency constraints between the branches to avoid unpleasant surprises at run-time, e.g. due to the system of equations all of a sudden becoming overdetermined. But this has not yet been investigated.

3.5 Implementation

The algorithm for computing the structural type for a signal relation has been prototyped in Haskell. It implements all aspects of the described algorithm, except that it has not been verified whether the computation of upper bounds indeed yields one of the least upper bounds. The time complexity of the algorithm is a concern. For example, the $\binom{m_M}{k}$ possible partitionings of the mixed equations that need to be investigated could, in adverse circumstances, be a large number. However, there may be ways to exploit more of the structure of the equations in order to limit the number of alternatives to consider. It is also easy to check how many partitioning there are before starting to enumerate them, and if they are judged to be too many, one can simply default to a safe over approximation of the type.

4 Structural types for a simple electrical circuit

As an example, let us apply the structural type system developed in section 3 to the simple electrical circuit from section 2.2.

Let us first consider the resistor. Recall that *Pin* is a record of two fields *v* and *i*, and that the signal relation interface thus consists of *four* variables: *p.v*, *p.i* , *n.v*, and *n.i*:

resistor :: *Resistance* → *SR* (*Pin*, *Pin*)
resistor r = **sigrel** (*p*, *n*) **where**
    twoPin ◊ (*p*, *n*, *u*)
    r * *p.i* = *u*

Before approximation, the two possible structural types for *resistor* are

$$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

reflecting a choice between using $u = p.v - n.v$ or $r * p.i = u$ for solving for the local variable *u*. (The equation $u = p.v - n.v$ is contributed by *twoPin*. However, note that only its *structural type* is of interest here, not the exact equation.) This gets approximated with a least upper bound to:

$$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

Of course, *resistor* cannot provide a single equation in which all of *p.v*, *p.i* , and *n.v* occur. But as the equation can only be used to solve for one of the variables, and as an equation can be provided for either two of the variables or the third, this is not too bad. Let us now consider *inductor* :

inductor :: *Inductance* → *SR* (*Pin*, *Pin*)
inductor l = **sigrel** (*p*, *n*) **where**
    twoPin ◊ (*p*, *n*, *u*)
    l * **der** *p.i* = *u*

The possible structural types before approximation are the same as for resistor, but this time reflecting a choice between using $u = p.v - n.v$ or $p.i = \int p.i' dt$, where *p.i* is the state derivative, for solving for the local variables. Note that the equation $l * p.i' = u$ is local, as neither the state derivative nor *u* occurs in the interface of *inductor*. After approximation, the structural type of *inductor* becomes the same as that of *resistor* . The case for *capacitor* is also very similar, and both the possible structural types prior to approximation and the final structural type are again the same. For a final example, suppose a mistake has been made in the definition of *simpleCircuit*: instead of

connect *r1n cp*
connect *r2n lp*

the equations read

connect *r2n lp*
connect *r2n lp*

Note that the number of equations and variables remain exactly the same in the two cases (each **connect** above is expanded to one equality constraint and one sum-to-zero equation). The structural type checking algorithm presented in this paper correctly reports that *simpleCircuit* is a locally underdetermined system. If only variables and equations had been counted, this error would not have been detected.

5 Future work

It should be emphasised that what has been presented in the present paper is only a preliminary investigation into the basics of a type-based structural analysis for modular systems of equations. It is not yet yet a full-featured type system. In particular, we have only considered the structural aspect in isolation, and to that end it was tacitly assumed that the structural types of composed signal relations were known, enabling the overall structural type of signal relations to be computed in a bottom-up manner.

However, FHM aims at treating signal relations as first class entities. One consequence of this is that signal relations can be *parametrised*, including on other signal relations. In FHM, a parametrised signal relation is simply a function that computes a signal relation given values of the parameters, which could include other signal relations. The question then is how to determine the structural type of any signal relation parameters.

One option would be to insist that the structural types of signal relation parameters is always declared. This could be cumbersome, but there is always the possibility of making a permissible (imprecise) default assumption in the absence of explicit declarations. Another option might be to try to infer suitable structural constraints for the parameters from how they are being used in Hindley-Milner fashion. A third option would be to move to a framework of *dependent types* [17, 16] where types are indexed by (can depend on) *terms*. In our case, the incidence matrices that represent the structural type would be considered term-level data, and the output structural type of a parametrised signal relation is then allowed to depend on the input structural type(s), or even the values of other parameters, meaning that the output structural type will be given as a function of the parameter values.

Incidentally, Modelica effectively also provides parametrised signal relations through its mechanism of replaceable components. Here the problem is addressed by syntactically requiring a default value for the replaceable component, which is used for type checking, and additionally insisting that any replacement conforms with the type of the default value in such a way that the result after any replacement is still guaranteed to be well-typed.

Another aspect that was not considered is how to handle equations on arrays. If the sizes of the arrays are manifestly known, it would be possible to consider an array equation simply as a shorthand notation for equations between the individual elements. But that is not very attractive, and it would inevitably lead to unwieldy structural types, bloated with lots of repetitive information. And, of course, if the array sizes are not manifest but parameters of the relation, it would be even more problematic. The most feasible approach is likely to restrict array equations in such a way that each such equation can be considered a single equation for the purpose of the structural types. Again, moving to a setting of dependent types might be helpful, as the type checking depends on term-level data, i.e. the sizes of the arrays. Dependent type systems supporting explicitly sized data has been studied extensively. One good example is Dependent ML [19, 20].

We would also like to integrate checking of physical dimensions [10] into the FHM type system. We observe that this is another reason to look closer at dependent types since the types become dependent on term-level data. For example, if an entity with a dimension type is subject to iterated multiplication, the resulting dimension depends on *how many times* the multiplication was iterated.

Finally, there are usability aspects that needs to be considered. While the type errors that are reported should be attributed fairly precisely to the component that is faulty, it is not clear how to phrase the error messages such that the problem becomes evident to the end user. Also, we need to keep in mind the conservative nature of the type system: there is no guarantee that further errors will not be discovered when a complete system of equations has been assembled. Combining the approach developed here with that of Bunus and Fritzson [3, 4] might help on both counts.

6 Conclusions

This paper presented a preliminary investigation into type system for modular systems of equations. The setting of the paper is equation-based, non-causal modelling, but the central ideas should have more general applicability. The paper showed how attributing a *structural type* to equation system fragments allows over- and underdetermined system fragments to be identified separately, without first having to assemble all fragments into a complete system of equations. The central difficulty was handling abstraction of systems of equations. The paper presented an algorithm for determining the best possible type for an abstracted system, although this may involve approximation.

It should be emphasised that was has been presented is not yet a complete type system. The paper only considers the structural aspect, and it was tacitly assumed that these structural types essentially could be determined in a straightforward bottom-up manner. The goal of treating signal relations as first class entities raises a number of further challenges, some of which were discussed in Section 5.

Acknowledgements

This work was supported by EPSRC grant EP/D064554/1. The author wishes to thank the anonymous referees for many useful suggestions that helped improve the paper.

References

[1] A.V. Aho, J.E. Hopcroft, J.D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[2] D. Broman, K. Nyström, P. Fritzson. *Determining over- and under-constrained systems of equations using structural constraint delta*. In GPCE '06: Proc. 5th Int. Conference on Generative Programming and Component Engineering, pp. 151–160, Portland, Oregon, USA, 2006. ACM.

[3] P. Bunus, P. Fritzson. *A debugging scheme for declarative equation based modeling languages*. In Proc. 4th Int. Symposium on Practical Aspects of Declarative Languages (PADL 2002), vol. 2257 of Lecture Notes in Computer Science, pages 280– 298, Portland, OR, USA, January 2002. Springer-Verlag.

[4] P. Bunus, P. Fritzson. *Methods for structural analysis and debugging of Modelica models*. In Proc. 2nd Int. Modelica Conference, pp. 157– 165, Oberpfaffenhofen, Germany, March 2002.

[5] A. Courtney, H. Nilsson, J. Peterson. *The Yampa arcade*. In Proc. 2003 ACM SIGPLAN Haskell Workshop (Haskell'03), pp. 7–18, Uppsala, Sweden, August 2003. ACM Press.

[6] A. L. Dulmage, N. S. Mendelsohn. *Coverings of bipartite graphs*. Canadian Journal of Mathematics, 10:517–534, 1958.

[7] H. Elmqvist. *A Structured Model Language for Large Continuous Systems*. PhD thesis TFRT-1015, Department of Automatic Control, Lund Institute of Technology, 1978.

[8] G. Giorgidze, H. Nilsson. *Switched-on Yampa: Declarative programming of modular synthesizers*. In P. Hudak, D.S. Warren, eds., Practical Aspects of Declarative Languages (PADL) 2008, vol. 4902 of Lecture Notes in Computer Science, pp. 282–298, San Francisco, CA, USA, January 2008. Springer-Verlag.

[9] P. Hudak, A. Courtney, H. Nilsson, J. Peterson. *Arrows, robots, and functional reactive programming*. In J. Jeuring, S.P. Jones, eds., Advanced Functional Programming, 4th International School 2002, vol. 2638 of Lecture Notes in Computer Science, pages 159–187. Springer-Verlag, 2003.

[10] A. Kennedy. *Dimension types*. In Proc. 5th European Symposium on Programming, no. 788 in Lecture Notes in Computer Science. Springer-Verlag, 1994.

[11] Modelica Association. *Modelica – A Unified Object--Oriented Language for Physical Systems Modeling: Tutorial* version 1.4, December 2000.

[12] Modelica Association. *Modelica – A Unified Object--Oriented Language for Physical Systems Modeling: Language Specification Version 3.0*, September 2007.

[13] H. Nilsson, A. Courtney, J. Peterson. *Functional reactive programming, continued*. In Proc. 2002 ACM SIGPLAN Haskell Workshop (Haskell'02), pp.51–64, Pittsburgh, Pennsylvania, USA, Oct. 02. ACM Press.

[14] H. Nilsson, J. Peterson, P. Hudak. *Functional hybrid modeling*. In Proc. PADL'03: 5th Int. Workshop on Practical Aspects of Declarative Languages, vol. 2562 of Lecture Notes in Computer Science, pages 376–390, New Orleans, Louisiana, USA, January 2003. Springer-Verlag.

[15] H. Nilsson, J. Peterson, P. Hudak. *Functional hybrid modeling from an object-oriented perspective*. In P. Fritzson, F. Cellier, and C. Nytsch-Geusen, eds., Proc. 1st Int. Workshop on Equation-Based Object-Oriented Languages and Tools, no. 24 in Linköping Electronic Conference Proceedings, pages 71–87. Linköping University Electronic Press, 2007.

[16] B.C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[17] S. Thompson. *Type Theory and Functional ProgramProgramming*. Addison-Wesley, 1991.

[18] Zhanyong W., P. Hudak. *Functional reactive programming from first principles*. In Proc. PLDI'01: Symposium on Programming Language Design and Implementation, pp. 242–252, June 2000.

[19] Hongwei Xi, F. Pfenning. *Eliminating array bound checking through dependent types*. In Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 249–257, Montreal, June 1998.

[20] Hongwei Xi, F. Pfenning. *Dependent types in practical programming*. In Proc. ACM SIGPLAN Symposium on Principles of Programming Languages, pages 214–227, San Antonio, January 1999.

Corresponding author: Henrik Nilsson,
School of Computer Science
University of Nottingham, United Kingdom,
nhn@cs.nott.ac.uk

Accepted EOOLT 2008, June 2008
Received: September 5, 2008
Accepted: November 29, 2008