Types in the Modelica Language

David Broman
Linköping University, Sweden
davbr@ida.liu.se

Peter Fritzson
Linköping University, Sweden
petfr@ida.liu.se

Sébastien Furic
Imagine, France

Abstract

Modelica is an object-oriented language designed for modeling and simulation of complex physical systems. To enable the possibility for an engineer to discover errors in a model, languages and compilers are making use of the concept of types and type checking. This paper gives an overview of the concept of types in the context of the Modelica language. Furthermore, a new concrete syntax for describing Modelica types is given as a starting point to formalize types in Modelica. Finally, it is concluded that the current state of the Modelica language specification is too informal and should in the long term be augmented by a formal definition.

Keywords: type system; types; Modelica; simulation; modeling; type safety

1 Introduction

One long term goal of modeling and simulation languages is to give engineers the possibility to discover modeling errors at an early stage, i.e., to discover problems in the model during design and not after simulation. This kind of verification is traditionally accomplished by the use of *types* in the language, where the process of checking for such errors by the compiler is called *type checking*. However, the concept of types is often not very well understood outside parts of the computer science community, which may result in misunderstandings when designing new languages. Why is then types important? Types in programming languages serve several important purposes such as naming of concepts, providing the compiler with information to ensure correct data manipulation, and enabling data abstraction. Almost all programming or modeling languages provide some kind of types. However, few language specifications include precise and formal definitions of types and type systems. This may result in incompatible compilers and unexpected behavior when using the language.

The purpose of this paper is twofold. The first part gives an overview of the concept of types, states concrete definitions, and explains how this relates to the Modelica language. Hence, the first goal is to augment the computer science perspective of language design among the individuals involved in the Modelica language design. The long-term objective of this work is to provide aids for further design considerations when developing, enhancing and simplifying the Modelica language. The intended audience is consequently engineers and computer scientists interested in the foundation of the Modelica language.

The second purpose and likewise the main contribution of this work is the definition of a concrete syntax for describing Modelica types. This syntax together with rules of its usage can be seen as a starting point to more formally describe the type concept in the Modelica language. To the best of our knowledge, no work has previously been done to formalize the type concept of the Modelica language.

The paper is structured as follows: Section 2 outlines the concept of types, subtypes, type systems and inheritance, and how these concepts are used in Modelica and other mainstream languages. Section 3 gives an overview of the three main forms of polymorphism, and how these concepts correlate with each other and the Modelica language. The language concepts and definitions introduced in Section 2 and 3 are necessary to understand the rest of the paper. Section 4 introduces the type concept of Modelica more formally, where we give a concrete syntax for expressing Modelica types. Finally, Section 5 state concluding remarks and propose future work in the area.

2 Types, Subtyping and Inheritance

There exist several models of representing types, where the *ideal model* [3] is one of the most well-known. In this model, there is a universe V of all values, containing all values of integers, real numbers, strings and data structures such as tuples, records and functions. Here, types are defined as sets of elements of the universe V. There is infinite number of types, but all types are not legal types in a programming language. All legal types holding some specific property, such as being an unsigned integer, are called *ideals*. Figure 1 gives an example of the universe V and two ideals: real type and function type, where the latter has the *domain* of integer and *codomain* of boolean.

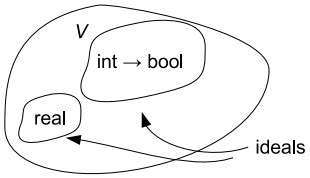

Figure 1: Schematic illustration of Universe V and two ideals.

In most mainstream languages, such as Java and C++, types are *explicitly typed* by stating information in the syntax. In other languages, such as Standard ML and Haskell, a large portion of types can be *inferred* by the compiler, i.e., the compiler deduces the type from the context. This process is referred to as *type inference* and such a language is said to be *implicitly typed*. Modelica is an explicitly typed language.

2.1 Language Safety and Type Systems

When a program is executed, or in the Modelica case: during simulation, different kinds of execution errors can take place. It is practical to distinguish between the following two types of runtime errors [2].

- *Untrapped errors* are errors that can go unnoticed and later cause arbitrary behavior of the system. For example, writing data out of bound of an array might not result in an immediate error, but the program might crash later during execution.
- *Trapped errors* are errors that force the computation to stop immediately; for example division by zero. The error can then be handled by the run-time system or by a language construct, such as exception handling.

A programming language is said to be *safe* if no untrapped errors are allowed to occur. These checks can be performed as *compile-time checks*, also called *static checks*, where the compiler finds the potential errors and reports them to the programmer. Some errors, such as array out of bound errors are hard to resolve statically. Therefore, most languages are also using *run-time checks*, also called *dynamic checking*. However, note that the distinction between compile-time and run-time becomes vaguer when the language is intended for interpretation.

Typed languages can enforce language safety by making sure that *well-typed* programs cannot cause type errors. Such a language is often called *type safe* or *strongly typed*. This checking process is called *type checking* and can be carried out both at run-time and compile-time.

The behavior of the types in a language is expressed in a *type system*. A type system can be described informally using plain English text, or formally using *type rules*. The Modelica language specification is using the former informal approach. Formal type rules have much in common with logical inference rules, and might at first glance seem complex, but are fairly straightforward once the basic concepts are understood. Consider the following:

$$\frac{\Gamma \vdash e_1 : \text{bool} \qquad \Gamma \vdash e_2 : T \qquad \Gamma \vdash e_3 : T}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T} \text{ (t-if)}$$

which illustrates a type rule for the following Modelica `if`-expression:

```
if e1 then e2 else e3
```

A type rule is written using a number of *premises* located above the horizontal line and a *conclusion* below the line. The *typing judgement* $\Gamma \vdash e : T$ means that expression e has type T with respect to a static typing environment Γ. Hence, the rule (t-if) states that *guard* e_1 must have the type of a boolean and that e_2 and e_3 must have the same type, which is also the resulting type of the `if`-expression after evaluation. This resulting type is stated in the last part of the conclusion, i.e., $: T$.

If the language is described formally, we can attempt to prove the *type soundness theorem* [15]. If the theorem holds, the type system is said to be *sound* and the language *type safe* or or just *safe*.

The concept of type safety can be illustrated by Robin Milner's famous statement "Well-typed programs cannot go wrong"[9]. Modern type soundness proofs are based on Wright and Felleisen's approach where type systems are proven correct together with the language's operational semantics [15]. Using this technique, informally stated, type safety hold if and only if the following two statements holds:

- *Preservation* - If an expression e has a type T and e evaluates to a value v, then v also has type T.

- *Progress* - If an expression e has a type T then either e evaluates to a new expression e' or e is a value. This means that a well typed program never gets "stuck", i.e., it cannot go into a undefined state where no further evaluations are possible.

Note that the above properties of type safety corresponds to our previous description of absence of untrapped errors. For example, if a division by zero error occurs, and the semantics for such event is undefined, the progress property will not hold, i.e., the evaluation gets "stuck", or enters an undefined state. However, if dynamic semantics are defined for throwing an exception when the division by zero operation occurs, the progress property holds.

For the imperative and functional parts of the Modelica language, the safety concept corresponds to the same methodology as other languages, such as Standard ML. However, for the instantiation process of models, the correspondence to the progress and preservation properties are not obvious.

Table 1 lists a number of programming languages and their properties of being type safe [10][2]. The table indicates if the languages are primarily designed to be checked statically at compile-time or dynamically at run-time. However, the languages stated to be statically type checked typically still perform some checking at runtime.

Although many of the languages are commonly believed to be safe, few have been formally proven to be so. Currently, ML [9] and subsets of the Java language [14] [7] has been proven to be safe.

Language	**Type Safe**	**Checking**
Standard ML	yes	static
Java	yes	static
Common LISP	yes	dynamic
Modelica	yes	static[1]
Pascal	almost	static
C/C++	no	static
Assembler	no	-

Table 1: Believed type safety of selected languages.

2.2 Subtyping

Subtyping is a fundamental language concept used in most modern programming languages. 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]. In this case, S is said to be a subtype of T, which is written as

$$S <: T \tag{1}$$

This relation can be described using the following important type rule called the *rule of subsumption.*

$$\frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T} \text{ (t-sub)}$$

The rule states that if $S <: T$, then every *term*[2] t of type S is also a term of type T. This shows a special form of *polymorphism*, which we will further explore in Section 3.

2.3 Inheritance

Inheritance is a fundamental language concept found in basically all class based *Object-Oriented (OO)* languages. From an existing *base class*, a new *subclass* can be created by *extending* from the base class, resulting in the subclass *inheriting* all properties from the base class. One of the main purposes with inheritance is to save programming

[1] One can argue whether Modelica is statically or dynamically checked, depending on how the terms compile-time and run-time are defined. Furthermore, since no exception handling is currently part of the language, semantics for handling dynamic errors such as array out of bound is not defined in the language and is therefore considered a compiler implementation issue.

[2] The word *term* is commonly used in the literature as an interchangeable name for expression.

and maintenance efforts of duplicating and reading duplicates of code. Inheritance can in principle be seen as an implicit code duplication which in some circumstances implies that the subclass becomes a subtype of the type of the base class.

Figure 2 shows an example[3] where inheritance is used in Modelica. A *model* called `Resistor` extends from a base class `TwoPin`, which includes two elements `v` for voltage and `i` for current. Furthermore, two instances `p` and `n` of connector `Pin` are public elements of `TwoPin`. Since `Resistor` extends from `TwoPin`, all elements `v`, `i`, `p` and `n` are "copied" to class `Resistor`. In this case, the type of `Resistor` will also be a subtype of `TwoPin`'s type.

```
connector Pin
  SI.Voltage v;
  flow SI.Current i;
end Pin;

partial model TwoPin
  SI.Voltage v;
  SI.Current i;
  Pin p, n;
equation
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end TwoPin;

model Resistor
  extends TwoPin;
  parameter SI.Resistance R=100;
equation
  R*i = v;
end Resistor;
```

Figure 2: Example of inheritance in Modelica, where a new subclass `Resistor` is created by extending the base class `TwoPin`.

However, a common misunderstanding is that subtyping and inheritance is the same concept [10]. A simple informal distinction is to say that "subtyping is a relation on interfaces", but "inheritance is a relation on implementations". In the resistor example, not only the public elements `v`, `i`, `p` and `n` will be part of class `Resistor`, but also the meaning of this class, i.e, the equations `v = p.v - n.v`, `0 = p.i + n.i` and `i = p.i`.

[3]These classes are available in the Modelica Standard Library 2.2, but are slightly modified for reasons of readability.

A famous example, originally stated by Alan Snyder [13], illustrates the difference between subtyping and inheritance. Three common *abstract data types* for storing data objects are *queue*, *stack* and *dequeue*. A queue normally has two operations, *insert* and *delete*, which stores and returns object in a *first-in-first-out (FIFO)* manner. A stack has the same operations, but are using a *last-in-first out (LIFO)* principle. A dequeue can operate as both a stack and a queue, and is normally implemented as a list, which allows inserts and removals at both the front and the end of the list.

Figure 3 shows two C++ classes modeling the properties of a dequeue and a stack. Since class `Dequeue` implements the properties also needed for a stack, it seems natural to create a subclass `Stack` that inherits the implementation from `Dequeue`. In C++, it is possible to use so called *private inheritance* to model inheritance with an *exclude operation*, i.e., to inherit some, but not all properties of a base class. In the example, the public methods `insFront`, `delFront`, and `delRear` in class `Dequeue` are inherited to be private in the subclass `Stack`. However, by adding new methods `insFront` and `delFront` in class `Stack`, we have created a subclass, which has the property of a stack by excluding the method `delRear`. `Stack` is obviously a subclass

```
class Dequeue{
public:
  void insFront(int e);
  int delFront();
  int delRear();
};

class Stack : private Dequeue{
public:
  void insFront(int e)
    {Dequeue::insFront(e);}
  int delFront()
    {return Dequeue::delFront();}
};
```

Figure 3: C++ example, where inheritance does not imply a subtype relationship.

of `Dequeue`, but is it a subtype? The answer is no, since an instance of `Stack` cannot be safely used when `Dequeue` is expected. In fact, the opposite is true, i.e., `Dequeue` is a subtype of `Stack` and not the other way around. However, in the following section we will see that C++ does not

treat such a subtype relationship as valid, but the type system of Modelica would do so.

2.4 Structural and Nominal Type Systems

During type checking, regardless if it takes place at compile-time or run-time, the type checking algorithm must control the relations between types to see if they are correct or not. Two of the most fundamental relations are *subtyping* and *type equivalence*.

Checking of type equivalence is the single most common operation during type checking. For example, in Modelica it is required that the left and right side of the equality in an equation have the same type, which is shown in the following type rule.

$$\frac{\Gamma \vdash e_1 : T \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 = e_2 : Unit} \text{ (t-equ)}$$

Note that type equivalence has nothing to do with equivalence of values, e.g., equation `4 = 10` is type correct, since integers 4 and 10 are type equivalent. However, this is of course not a valid equation, since the values on the right and left side are not the same.

The *Unit* type (not to confuse with physical units), shown as the resulting type of the equation, is often used as a type for uninteresting result values.

A closely related concept to type equivalence is *type declaration*, i.e., when a type is declared as a specific *name* or *identifier*. For example, the following Modelica record declaration

```
record Person
  String name;
  Integer age;
end Person;
```

declares a type with name `Person`. Some languages would treat this as a new unique type that is not equal to any other type. This is called *opaque* type declaration. In other languages, this declaration would simply mean that an alternative name is given to this type. However, the type can also be expressed by other names or without any name. This latter concept is commonly referred as *transparent* type declaration.

In a pure *nominal type system*, types are compared (subtyping and type equivalence) by using the *names* of the declared types, i.e., opaque type declarations are used. Type equivalence is controlled by checking that the same declared name is used. Furthermore, the subtype relation in OO languages is checked by validating the inheritance order between classes. The C++ language is mainly using a nominal type system, even if parts of the language does not obey the strict nominal structure.

Consider the listing in Figure 4, which illustrates a C++ model similar to the resistor example earlier given as Modelica code in Figure 2. In this case, `Resistor` is a subclass of `TwoPin` and the type of `Resistor` is therefore also a subtype of `TwoPin`'s type. However, the type of `Inductor` is not a subtype to the type of `TwoPin`, since `Inductor` does not inherit from `TwoPin`. Moreover, `Resistor2` is not type equivalent to `Resistor` even if they have the same structure and inherits from the same base class, since they are opaquely declared.

```
class Pin{
public:
  float v, i;
};

class TwoPin{
public:
  TwoPin() : v(0),i(0){};
  float v, i;
  Pin p, n;
};

class Resistor : public TwoPin{
public:
  Resistor() : r(100) {};
  float r;
};

class Resistor2 : public TwoPin{
public:
  Resistor() : r(200) {};
  float r;
};

class Inductor{
public:
  TwoPin() : v(0),i(0){};
  float v, i;
  Pin p, n;
  const float L;
};
```

Figure 4: `Resistor` inheritance example in C++.

In a *structural type system* [12], declarations are introducing new names for type expressions, but no new types are created. Type equivalence and subtype relationship is only decided depending on the structure of the type, not the naming.

The Modelica language is inspired by the type system described by Abadi and Cardelli in [1] and is using transparent type declarations, i.e., Modelica has a structural type system. Consider the `Resistor` example given in Figure 2 and the two complementary models `Inductor` and `Resistor2` in Figure 5. Here, the same relations holds between `TwoPin` and `Resistor`, i.e., that the type of `Resistor` is a subtype of `TwoPin`'s type. The same holds between `TwoPin` and `Resistor2`. However, now `Resistor` and `Resistor2` are type equivalent, since they have the same structure and naming of their public elements. Furthermore, the type of `Inductor` is now a valid subtype of `TwoPin`'s type, since `Inductor` contains all public elements (type and name) of the one available in `TwoPin`.

```
model Resistor2
  extends TwoPin;
  parameter SI.Resistance R=200;
equation
  R*i = v;
end Resistor;

model Inductor
  Pin p, n;
  SI.Voltage v;
  SI.Current i;
  parameter SI.Inductance L=1;
equation
  L*der(i) = v;
end Inductor;
```

Figure 5: Complementary `Inductor` and `Resistor2` models to the example in Figure 2.

It is important to stress that *classes* and *types* in a structural type system are **not** the same thing, which also holds for Modelica. The type of a class represents the interface of the class relevant to the language's type rules. The type does not include implementation details, such as equations and algorithms.

Note that a nominal type system is more restrictive than a structural type system, i.e., two types that have a structured subtype relation can always have a subtype relation by names (if the language's semantics allows it). However, the opposite is not always true. Recall the `Dequeue` example listed in Figure 3. The class `Stack` has a subclass relation to `Dequeue`, but a subtype relation cannot be enforced, due to the structure of the class. The converse could be true, but the type system of C++ would not allow it, since it is nominal and subtype relationships are based on names. Hence, a structural type system can be seen as more *expressive* and *flexible* compared to a nominal one, even if both gives the same level of language type safety.

3 Polymorphism

A type system can be *monomorphic* in which each value can belong to at most one type. A type system, as illustrated in Figure 1, consisting of the distinct types function, integer, real, and boolean is a monomorphic type system. Conversely, in a *polymorphic* type system, each value can belong to many different types. Languages supporting polymorphism are in general more expressive compared to languages only supporting monomorphic types. The concept of polymorphism can be handled in various forms and have different naming depending on the paradigm where it is used. Following John C. Mitchell's categorization, polymorphism can be divided into the following three main categories [10]:

- Subtype Polymorphism
- Parametric Polymorphism
- Ad-hoc Polymorphism

There are other similar categorizations, such as Cardelli and Wegner's [3], where the ad-hoc category is divided into *overloading* and *coercion* at the top level of categories.

3.1 Subtype Polymorphism

Subtyping is an obvious way that gives polymorphic behavior in a language. For example, an instance of `Resistor` can be represented both as an `TwoPin` type and a `Resistor` type. This statement can also be shown according to the rule of subsumption (t-sub) described in Section 2.2.

When a value is changed from one type to some supertype, it is said to be an *up-cast*. Up-casts can be viewed as a form of *abstraction* or *information hiding*, where parts of the value becomes

invisible to the context. For example, an up-cast from `Resistor`'s type to `TwoPin`'s type hides the parameter `R`. Up-casts are always type safe, i.e., the run-time behavior cannot change due to the upcast.

However, for subtype polymorphism to be useful, typically types should be possible to *down-cast*, i.e., to change to a subtype of a type's value. Consider function `Foo`

```
function Foo
  input TwoPin x;
  output TwoPin y;
end Foo;
```

where we assume that down-casting is allowed[4]. It is in this case valid to pass either a value of type `TwoPin` (type equivalence) or a subtype to the type of `TwoPin`. Regardless if a value of `TwoPin`'s or `Inductor`'s type is sent as input to the function, a value of `TwoPin`'s type will be returned. It is not possible for the static type system to know if this is a `TwoPin`, `Resistor` or a `Inductor` type. However, for the user of the function, it might be crucial to handle it as an `Inductor`, which is why a down-cast is necessary.

Down-casting is however not a safe operation, since it might cast down to the wrong subtype. In Java, before version 1.5 when *generics* were introduced, this safety issue was handled using dynamic checks and raising dynamic exceptions if an illegal down-cast was made. Subtype polymorphism is sometimes called "poor-man's polymorphism", since it enables polymorphic behavior, but the safety of down-casts must be handled dynamically [12].

The Modelica language supports subtyping as explained previously, but does not have any operation for down-cast. Since the language does not include this unsafe operation, only a limited form of subtype polymorphism can be used with functions. For example, a function can operate on a polymorphic type as input, such as `TwoPin`, but it only makes sense to return values of a type that can be instantly used by the caller.

However, subtype polymorphism is more extensively used when reusing and replacing components in models, i.e., by using the `redeclare` keyword.

[4]This function type or example is not valid in the current Modelica standard. It is used only for the purpose of demonstrating subtype polymorphism.

3.2 Parametric Polymorphism

The term *parametric polymorphism* means that functions or classes can have *type parameters*, to which types or *type expressions* can be supplied. The term parametric polymorphism is often used in functional language communities, while people related to object-oriented languages tend to use the term *generics*.

The C++ *template* mechanism is an example of *explicit parametric polymorphism*, where the type parameter must be explicitly declared. Consider for example Figure 6, where a template function `swap` is implemented. The type parameter `T` must be explicitly stated when declaring the function. However, the type argument is not needed when calling the function, e.g., both `int x,y; swap(x,y);` and `float i,j; swap(i,j)` are valid usage of the function.

```
template<typename T>
void swap(T& x, T& y){
  T tmp = x;
  x = y;
  y = tmp;
}
```

Figure 6: Explicit parametric polymorphism in C++.

Standard ML on the other hand is making use of *implicit parametric polymorphism*, where the type parameters do not need to be explicitly stated when declaring the function. Instead, the *type inference algorithm* computes when type parameters are needed.

A notable difference of parametric and subtype polymorphism is that all type checking of parametric polymorphism can take place at compile-time and no unsafe down-cast operation is needed.

Standard ML and and C++ are internally handling parametric polymorphism quite differently. In C++ templates, instantiation to compiled code of a function is done at link time. If for example function `swap` is called both using `int` and `float`, different code of the function is generated for the two function calls. Standard ML on the other hand is using *uniform data representation*, where all data objects are represented internally as pointers/references to objects. Therefore, there is no need to create different copies of code for different types of arguments.

Modelica can be seen to support a limited version of parametric polymorphism, by using the *redeclare* construct on local class declarations.

3.3 Ad-hoc Polymorphism

In parametric polymorphism the purpose is to declare one implementation that can be used with different types of arguments. *Ad-hoc polymorphism*, by contrast, allows a polymorphic value to be used differently depending on which type the value is viewed to have.

There are several language concepts that fall under the concept of ad-hoc polymorphism [3], where *Overloading* and *Coercion* are most notable. Other related concepts that also fall under this category are Java's `instanceOf` concept and different form of *pattern matching* [12].

3.3.1 Overloading

A symbol is *overloaded* if it has two or more meanings, which are distinguished by using types. That is, a single function symbol or identifier is associated with several implementations.

An example of overloading that exists in many programming languages is *operator overloading* for built in types. For example, the symbol + is using infix notation and have two operands associated with it. The type of these operands decide how the operation should be carried out, i.e., which implementation that should be used.

Overloading can take place at either compile-time or at run-time. Overloading used at run-time is often referred to as *dynamic lookup*[10], *dynamic dispatch* or *multi-method dispatch*. In most cases, the single term overloading refers to static overloading taking place at compile-time. The distinction becomes of course vague, if the language is *interpreted* and not compiled.

Another form of overloading available in some languages is user-defined *function overloading*, where a function identifier can represent several implementations for different type arguments. Modelica is currently not supporting any form of user defined overloading.

3.3.2 Coercion

Another form of ad-hoc polymorphism is *coercion* or *implicit type conversion*, which is run-time conversion between types, typically performed by code automatically inserted by the compiler. The distinction between overloading and type coercion is not always clear, and the two concepts are strongly related. Consider the following four expressions of multiplication [3]:

```
7   * 9   //Integer * Integer
6.0 * 9.1 //Real * Real
6   * 5.2 //Integer * Real
6.0 * 8   //Real * Integer
```

All four of these expressions are valid Modelica expressions, but they can in the context of coercion and overloading be interpreted in three different ways:

- The multiplication operator is overloaded four times, one for each of the four expressions.
- The operator is overloaded twice; one for each of the the first two expressions. If the arguments have different types, i.e., one is `Real` and the other one `Integer`, type coercion is first performed to convert the arguments to `Real`.
- Arguments are always implicitly converted to `Real`, and the operator is only defined for `Real`s.

Type conversions can also be made *explicit*, i.e., code is inserted manually by the programmer that converts the expression to the correct type.

In Modelica, implicit type conversion is used when converting from `Integer` to `Real`. Of the three different cases listed above, the second one applies to the current Modelica 2.2 standard.

4 Modelica Types

In the previous sections we described different aspects of types for various languages. In this section we will present a concrete syntax for describing Modelica types, followed by rules stating legal type expressions for the language.

The current Modelica language specification [11] specifies a formal syntax of the language, but the semantics including the type system are given informally using plain English. There is no explicit definition of the type system, but an implicit description can be derived by reading the text describing relations between types and classes in the Modelica specification. This kind of implicit specification makes the actual specification open for interpretation, which may result in incompatible

compilers; both between each other, but also to the specification itself. Our work in this section should be seen as a first step to formalize what a type in Modelica actually is. Previous work has been performed to formally specify the semantics of the language [8], but without the aim to more precisely define the exact meaning of a type in the language.

Why is it then so important to have a precise definition of the types in a language? As we have described earlier, a type can be seen as an interface to a class or an object. The concept of interfaces forms the basis for the widely accepted approach of separating *specification* from *implementation*, which is particularly important in large scale development projects. To put it in a Modelica modeling context, let us consider a modeling project of a car, where different modeling teams are working on the wheels, gearbox and the engine. Each team has committed to provide a set of specific attributes for their component, which specifies the interface. The contract between the teams is not violated, as long as the individual teams are following this commitment of interface (the specification) by adding / removing equations (the implementation). Since the types state the interfaces in a language with a structural type system, such as Modelica, it is obviously decisive that they have a precise definition.

Our aim here is to define a precise notation of types for a subset of the Modelica language, which can then further be extended to the whole language. Since the Modelica language specification is open for interpretation, the presented type definition is our interpretation of the specification.

4.1 Concrete Syntax of Types

Now, let us study the types of some concrete Modelica models. Consider the following model `B`, which is rather uninteresting from a physical point of view, but demonstrates some key concepts regarding types.

```
model B
  parameter Real s=-0.5;
  connector C
    flow Real p;
    Real q;
  end C;
protected
  Real x(start=1);
equation
  der(x) = s*x;
end B;
```

What is the type of model `B`? Furthermore, if `B` was used and instantiated as a component in another model, e.g., `B b;`, what would the resulting type for element `b` be? Would the type for `B` and `b` be the same? The answer to the last question is definitely no. Consider the following listing, which illustrates the type of model `B`.

```
model classtype //Class type of model B
  public parameter Real objtype s;
  public connector classtype
    flow Real objtype p;
    nonflow Real objtype q;
  end C;
  protected Real objtype x;
end
```

This type listing follows the grammar syntax listed in Figure 7. The first thing to notice is that the name of model `B` is not visible in the type. Recall that Modelica is using a structural type system, where the types are determined by the structure and not the names, i.e., the type of model `B` has nothing to do with the name `B`. However, the names of the *elements* in a type are part of the type, as we can see for parameter `s` and variable `x`.

The second thing to observe is that the equation part of the model is missing in the type definition. The reason for this is that equations and algorithms are part of the implementation and not the model interface. Moreover, all elements `s`, `C` and `x` are preserved in the type, but the keywords `model`, `connector` and basic type `Real` are followed by new keywords `classtype` or `objtype`. This is one of the most important observations to make regarding types in a class based system using structural subtyping and type equivalence. As we can see in the example, the type of model `B` is a *class type*, but parameter `s` is an *object type*. Simply stated: A class type is the type of one of Modelica's restricted classes, such as `model`, `connector`, `record` etc., but an *object type* is the type of an instance of a class, i.e., an object. Now, the following shows the object type of `b`, where `b` represents an instance of model `B`:

```
model objtype //Object type of b
  parameter Real objtype s;
end
```

Obviously, both the type of connector `C` and variable `x` have been removed from the type of `b`. The reason is that an object is a run-time entity, where neither local classes (connector `C`) nor protected elements (variable `x`) are accessible from

outside the instance. However, note that this is not the same as that variable `x` does not exist in a instance of `B`; it only means that it is not visible to the outside world.

Now, the following basic distinctions can be made between *class types* and *object types*:

- Classes can inherit (using extends) from class types, i.e., the type that is bound to the name used in an `extends` clause must be a class type and not an object type.
- Class types can contain both object types and class types, but object types can only hold other object types.
- Class types can contain types of protected elements; object types cannot.
- Class types are used for compile time evaluation, such as inheritance and redeclarations.

type	::=	(**model** | **record** | **connector** | **block** | **function** | **package**) *kindoftype* {{*prefix*} *type identifier* **;**} **end**
	|	(**Real** | **Integer** | **Boolean** | **String**) *kindoftype*
	|	**enumeration** *kindoftype enumlist*
kindoftype	::=	**classtype** | **objtype**
prefix	::=	*access* | *causality* | *flowprefix* | *modifiability* | *variability* | *outerinner*
enumlist	::=	**(** *identifier* {**,** *identifier*} **)**
access	::=	**public** | **protected**
causality	::=	**input** | **output** | **inputoutput**
flowprefix	::=	**flow** | **nonflow**
modifiability	::=	**replaceable** | **modifiable** | **final**
variability	::=	**constant** | **parameter** | **discrete** | **continuous**
outerinner	::=	**outer** | **inner** | **notouterinner**

Figure 7: Concrete syntax of partial Modelica types.

Let us now take a closer look at the grammar listed in Figure 7. The root non-terminal of the grammar is *type*, which can form a class or object type of the restricted classes or the built in types `Real`, `Integer`, `Boolean`, `String`, or `enumeration`. The grammar is given using a variant of *Extended Backus-Naur Form* (EBNF), where terms enclosed in brackets {} denote zero, one or more repetitions. Keywords appearing in the concrete syntax are given in bold font. All prefixes, such as `public`, `flow`, `outer` etc. can be given infinitely many times. The correct usage of these prefixes is not enforced by the grammar, and must therefore be handled later in the semantic analysis. We will give guidelines for default prefixes and restrictions of the usage of prefixes in the next subsection.

Now, let us introduce another model `A`, which extends from model `B`:

```
model A
  extends B(s=4);
  C c1;
equation
 c1.q = -10*der(x);
end A;
```

The question is now what the type of model `A` is and if it is instantiated to an object, i.e., `A a;`, what is then the type of `a`? The following shows the type of model `A`.

```
model classtype //Class type of A
  public parameter Real objtype s;
  public connector classtype
    flow Real objtype p;
    nonflow Real objtype q;
  end C;
  public connector objtype
    flow Real objtype p;
    nonflow Real objtype q;
  end c1;
  protected Real objtype x;
end
```

First of all, we see that the type of model `A` does not include any `extends` keyword referring to the inherited model `B`. Since Modelica has a structural type system, it is the structure that is interesting, and thus a type only contains the collapsed structure of inherited elements. Furthermore, we can see that the protected elements from `B` are still available, i.e., inheritance preserves the protected element after inheritance. Moreover, since model `A` contains an instance of connector `C`, this is now available as an object type for element `c1` in the class type of `A`. Finally, consider the type of an

instance a of class A:

```
model objtype //Object type of a
  parameter Real objtype s;
  connector objtype
    flow Real objtype p;
    nonflow Real objtype q;
  end c1;
end
```

The protected element is now gone, along with the elements representing class types. A careful reader might have noticed that each type definition ends without a semi-colon, but elements defined inside a type such as `model classtype` ends with a semi-colon. A closer look at the grammar should make it clear that types themselves do not have names, but when part of an element definition, the type is followed by a name and a semi-colon. If type expressions were to be ended with a semi-colon, this recursive form of defining concrete types would not be possible.

4.2 Prefixes in Types

Elements of a Modelica class can be prefixed with different notations, such as `public`, `outer` or `replaceable`. We do not intend to describe the semantics of these prefixes here, instead we refer to the specification [11] and to the more accessible description in [5]. Most of the languages prefixes have been introduced in the grammar in Figure 7. However, not all prefixes are allowed or have any semantic meaning in all contexts.

In this subsection, we present a partial definition of when different prefixes are allowed to appear in a type. In currently available tools for Modelica, such as Dymola [4] and OpenModelica [6], the enforcement of these restrictions is sparse. The reason for this can both be the difficulties to extract this information from the specification and the fact that the rules for the type prefixes are very complex.

In Figure 8 several abbreviations are listed. The lower case abbreviations a, c, c' etc. define sets of prefixes. The uppercase abbreviations M, R etc. together with a subscription of c for class type and o for object type, represents the type of an element part of another type. For example M_c is a model class type, and R_o is a record object type.

Now, consider the rules for allowed prefixes of elements shown in the tables given in Figure 9, Figure 10, and Figure 11.

In Figure 9 the intersection between the column (the type of an element) and the row (the type that contains this element) states the allowed prefixes for this particular element. This table shows which prefixes that are allowed for a class type that is part of another class type. For example, recall the connector `C` in model `A`. When looking at the type of `A`, we have a class type (the model class type) that contains an-

$M =$	**model**		
$R =$	**record**		
$C =$	**connector**		
$B =$	**block**		
$F =$	**function**		
$P =$	**package**		
$X =$	**Integer**, **Boolean**, **enumeration**, **String**		
$Y =$	**Real**		
$a =$	{**public**, **protected**}	Access	
$a' =$	{**public**}		
$c =$	{**input**, **output**, **inputoutput**}	Causality	
$c' =$	{**input**, **output**}		
$f =$	{**flow**, **nonflow**}	Flowprefix	
$m =$	{**replaceable**, **modifiable**, **final**}	Modifiability	
$m' =$	{**modifiable**, **final**}		
$v =$	{**constant**, **parameter** **discrete**, **continuous**}	Variability	
$v' =$	{**constant**, **parameter** **discrete**}		
$v'' =$	{**constant**}		
$o =$	{**outer**, **inner**, **notouterinner**}	Outerinner	

Figure 8: Abbreviation for describing allowed prefixes. Default prefixes are underlined.

	M_c	R_c	C_c	B_c	F_c	P_c	X_c	Y_c
M_c	amo	amo	amo	amo	amo	.	amo	amo
R_c	.	.	.	.	.	.	.	.
C_c	.	.	.	.	.	.	.	.
B_c	amo	amo	amo	amo	amo	.	amo	amo
F_c	.	am	.	.	am	.	am	am
P_c	am	amv''	am	am	am	$a'm$	am	am

Figure 9: Prefixes allowed for elements of class type (columns) inside a class type (rows).

	M_o	R_o	C_o	B_o	F_o	P_o	X_o	Y_o
M_c	amo	$acmo$	$acmo$	amo	amo	.	$acmv'o$	$acmvo$
R_c	.	mo	.	.	.	.	$mv'o$	mvo
C_c	.	mo	mo	.	.	.	m	$mcfvo$
B_c	amo	$ac'mo$	$ac'mo$	amo	amo	.	$ac'mv'o$	$ac'mvo$
F_c	.	$ac'm$	.	.	am	.	$ac'mv'$	$ac'mv$
P_c	.	amv''	.	.	.	.	amv''	amv''

Figure 10: Prefixes allowed for elements of object type (columns) inside a class type (rows).

	M_o	R_o	C_o	B_o	F_o	P_o	X_o	Y_o
M_o	o	$cm'o$	co	o	o	.	$cm'v'o$	$cm'vo$
R_o	.	$m'o$	.	.	.	.	$m'v'o$	$m'vo$
C_o	.	$m'o$	o	.	.	.	.	$cfm'vo$
B_o	o	$c'o$	$c'o$	o	o	.	$c'm'v'o$	$c'm'vo$
F_o	.	c'	.	.	.	.	$m'v'$	$m'v$
P_o	.	.	.	.	.	.	.	.

Figure 11: Prefixes allowed for elements of object type (columns) inside an object type (rows).

other class type (the connector class type), i.e., the allowed prefixes are given in the intersection of row 1 and column 3. In this case, *access* prefixes `public` and `protected`, *modifiability* prefixes `replaceable`, `modifiable`, and `final`, and *outer/inner* prefixes `outer`, `inner` and `notouterinner` are allowed.

We have introduced a number of new prefixes: `inputoutput`, `notouterinner`, `nonflow`, `modifiable`, and `continuous`. These new prefixes are introduced to enable a complete type definition, e.g., it should be possible to explicitly specify that a variable in a connector is not a flow variable by giving a `nonflow` prefix. However, for simplicity, sometimes it is more convenient to leave out some of the prefixes, and instead use default prefixes. The defined default prefixes are show underlined in Figure 8. If no underlined prefix exists in a specific set, this implies that the prefix must be explicitly stated.

Analogous to the description of Figure 9, Figure 10 shows the allowed prefixes for elements of object types contained in a class type and Figure 11 shows object types contained in object types. There are no tables given for class types contained in object types for the simple reason that object types are not allowed to contain class types.

In some of the cells in the tables described above, a dot symbol is shown. This means that the specific type of element inside a certain type is not allowed. Hence, such a combination should not be allowed by the compiler at compile-time.

Now, let us observe some general trends between the allowed attributes. First of all, object types cannot contain class types, which is why there are only 3 tables. Secondly, access prefixes (`public`, `protected`) are only allowed in class types, which is why Figure 11 does not contain any abbreviation a. Thirdly, the `replaceable` prefix does not make sense in object types, since redeclarations may only occur during object creation or inheritance, i.e., compile-time evaluation. Then when an object exists, the type information for replaceable is of no interest any more. Finally, we can see that package class types can hold any other class types, but no other class type can hold package types.

Note that several aspects described here are our design suggestions for simplifying and making the language more stringent from a type perspective. Currently, there are no limitations for any class to contain packages in the Modelica specification. Furthermore, there are no strict distinctions between object- and class types, since elaboration and type checking are not clearly distinguished. Hence, redeclaration of elements in an object are in fact possible according to the current specification, even if it does not make sense in a class based type perspective.

4.3 Completeness of the Type Syntax

One might ask if this type definition is complete and includes all aspects of the Modelica language and the answer to that question is no. There are several aspects, such as arrays, partial and encapsulated classes, units, constrained types, conditional components and external functions that are left out on purpose.

The main reason for this work is to pinpoint the main structure of types in Modelica, not to formulate a complete type definition. As we can see from the previous sections, the type concept in the language is very complex and hard to define, due to the large number of exceptions and the informal description of the semantics and type system in the language specification.

The completeness and correctness of the allowed type prefixes described in the previous section depend on how the specification is interpreted. However, the notation and structure of the concrete type syntax should be consistent and is intended to form the basis for incorporating this improved type concept tighter into the language.

Finally, we would like to stress that defining types of a language should be done in parallel with the definition of precise semantic and type rules. Since the latter information is currently not available, the precise type definition is obviously not possible to validate.

5 Conclusion

We have in this paper given a brief overview of the concept of types and how they relate to the Modelica language. The first part of the paper described types in general, and the latter sections detailed a syntax definition of how types can be expressed for the Modelica language.

The current Modelica specification uses *Extended Backus-Naur Form* (EBNF) for specifying the syntax, but the semantics and the type system are informally described. Moreover, the Modelica language has become difficult to reason about, since it has grown to be fairly large and complex. By giving the types for part of the language we have illustrated that the type concept is complex in the Modelica language, and that it is non-trivial to extract this information from the language specification.

Consequently, we think that it is important to augment the language specification by using more formal techniques to describe the semantics and the type system. We therefore propose that a subset of Modelica should be defined, which models the core concepts of the language. This subset should be describe using operational semantics including formal type rules. For some time, denotational semantics has been used as the semantic language of choice, however it has been shown to be less cumbersome to prove type soundness using operational semantics [15].

In the short term, this proposed core language is supposed to be used as basic data for better design decision-making, not as an alternative or replacement of the official Modelica specification. However, the long term goal should, in our option, be to describe the entire Modelica language formally.

Acknowledgments

Thanks to Thomas Schön and Kaj Nyström for many useful comments of this paper.

This research work was funded by CUGS (the Swedish National Graduate School in Computer Science), by SSF under the VISIMOD project, and by Vinnova under the NETPROG Safe and Secure Modeling and Simulation on the GRID project.

References

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, New York, USA, 1996.

[2] Luca Cardelli. Type Systems. In *The Computer Science and Engineering Handbook*, chapter 97. CRC Press, second edition, 2004.

[3] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Comput. Surv.*, 17(4):471–523, 1985.

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

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

[6] 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, 2005.

[7] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[8] David Kågedal and Peter Fritzson. Generating a Modelica Compiler from Natural Semantics Specifications. In *Proceedings of the Summer Computer Simulation Conference*, 1998.

[9] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[10] John C. Mitchell and Krzysztof Apt. *Concepts in Programming Languages*. Cambridge University Press, 2003.

[11] 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].

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

[13] Alan Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 38–45, New York, USA, 1986. ACM Press.

[14] Don Syme. Proving Java Type Soundness. *Lecture Notes in Computer Science*, 1523:83, 1999.

[15] Andrew K. Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115(1):38–94, 1994.