# On the Foundation of K: Data types versus Relations

Klaus Havelund

Jet Propulsion Laboratory
California Institute of Technology
California, USA

## 1 Introduction

This document discusses a conflict in the K language that I believe we are confronted with. In short: we are trying to make two views co-exist, what I will call:

1. the *data type* view
2. the *relational* view

The document first in Section 2 attempts to explain the two views. Then in Section 3 addresses a problem in how we approach associations using the data type view, and fixing it by making it “more relational”. In Section 4 we show how this fix causes yet a problem in definition of classes, where we so far also have adopted the data type view. This leads to the main question: *do we also need to shift to the relational view in classes?* How relational should we be? Section 5 offers a proposal/solution. Finally, Section 6 concludes the paper.

## 2 The two views

### 2.1 The data type view

Formal specification languages such as VDM, Z, and most/all programming languages, such as Java and Scala, are based on the data type view. These languages have composite data types such as records, sets, lists, maps, etc. K is based on this view as well. In these languages objects have structure. For example, in Scala we can write:

```
class Person {
  var cars : Set[Car]
}
```

This means that a *Person* object $p$ is a composite structure, which includes a reference to a set of cars. $p.cars$ denotes that set by returning a direct reference[1] to it, namely the reference (pointer) contained in $p$. We have adopted this view

[1] Whether it is a reference or the set itself is not important.

in K since this is what we are most comfortable with in our daily programming activities, and since it is the most common view adopted in formal specification languages, except one: ALLOY, which is discussed below. In K the above class becomes (we can discuss whether it is better with a Set name or **set** keyword, but that is another discussion):

```
class Person {
  var cars : {Car}
}
```

This is meant to have the same semantics as in SCALA: a *Person* object *p* contains an internal reference to a set of cars obtained by *p.cars*.

Mathematically, we can say that an object is a finite mapping from names (the property names) to values, and values include for example sets, that is: structured objects:

$$Object = PropertyName \overset{m}{\to} Value \tag{1}$$

$$Value = Integer \cup Boolean \cup \ldots \cup Set[Value] \cup List[Value] \tag{2}$$

As an example, consider an object $p$ where $p.cars = \{c_1, c_2\}$. This object will have the form:

$$p = [cars \mapsto \{c_1, c_2\}]$$

In K we can in addition constrain a set with multiplicities. That is, 'Car 0..10' is a type expression representing the type of sets of cars that have a size no bigger than 10. With this type expression we can hence constrain our K specification further as follows:

```
class Person {
  var cars : {Car} 0..10
}
```

This really is a shorthand for using a predicate subtype expression, as in:

```
class Person {
  var cars : {|carset : {Car} | carset.size() <= 10|}
}
```

At this point you may be lost, but that is not crucial for understanding the remainder of the document. You probably got the gist of it.

## 2.2 The relational view

In the relational view, on the other hand, there are only atoms, which has **no structure**, and relations (sets of tuples) between atoms. This can indeed be considered as the foundation of UML/SYSML block diagrams, although it is not quite clear since no formal semantics of UML/SYSML seems to have been agreed upon. However, it is fair to say that UML/SYSML block diagrams strongly emphasize a relational view.

This relational view is also adopted in the ALLOY specification language, which was designed inspired by UML and Z. ALLOY was designed for specification and analysis, but in contrast to languages such as VDM and Z, automated analysis was prioritized. In ALLOY we can write the following specification:

```
sig Person {
  cars : set Car
}
```

Here **sig** stands for *signature* but may be read as **class** at an informal level. However, this means something quite different than in the SCALA and K (unless we change that) case above. The above specification states that there are two sets $Person$ and $Car$, containing atoms which have **no** structure. Furthermore, there is a relation between $Person$ and $Car$ that associates any person $p$ in $Person$ with zero or more cars.

Mathematically this means that we are defining two sets of unstructured atoms:

$$Person, Car$$

and then a relation between them:

$$cars \subseteq Person \times Car$$

That is, $cars$ is a set of tuples of the form $(p, c)$ indicating that person $p$ is related to car $c$. As an example, consider an object $p$ related to the cars $c_1$ and $c_2$. This means that we have the sets and the relation:

$$Person = \{p\} \tag{3}$$
$$Car = \{c_1, c_2\} \tag{4}$$
$$cars = \{(p, c_1), (p, c_2)\} \tag{5}$$

Note that we can still write $p.cars$ (as in OCL), and this expression will return the set:

$$p.cars = \{c : Car \mid (p, c) \in cars\}$$

## 3 Associations in K

In this section we will have a look at how we sofar have planned to handle associations in K as separate classes, so-called association classes. I will point out a problem with this approach, and will suggest an alternative, which however then leads to yet another problem to be tackled in the following section.

### 3.1 The problem with the current K associations

Note that in all programs/specifications above, the relation between persons and cars have been expressed as belonging to $Person$. It is defined as a property in class $Person$. Here we instead try to extract such relations in association classes. The above example could in K instead be written as follows (ignoring the multiplicity 0..10):

```
class Person {}
class Car {}

assoc cars {
  var person : Person;
  var car : {Car}
}
```

Note that the property name $cars$ denotes a **set of cars**. This is the data type view. An instance of the $cars$ class (relation) is a structure of the following form for our example above:

$$[person \mapsto p, car \mapsto \{c_1, c_2\}]$$

That is, it maps a single person $p$ to all the cars it is related to. This view, however, leads to a problem in my opinion. Assume that we start with an empty relation and that we now execute a statement to connect $p$ with $c_1$" (syntax not quite settled on):

```
cars.connect(p,c1);
```

After this statement we will have the structure:

$$[person \mapsto p, car \mapsto \{c_1\}]$$

Assume now that we connect $p$ with $c_2$:

```
cars.connect(p,c2);
```

After this statement we will have the structure:

$$[person \mapsto p, car \mapsto \{c_1, c_2\}]$$

However, adding the connection is complicated. We have to find the $R$ instance it belongs to, namely the one identified by $p$. I find this to be an undesirable complication.

### 3.2 Fixing our associations

A simpler approach would be to follow the relational view in this case, and to define our association as follows:

```
assoc cars {
  var person : Person;
  var car : Car
}
```

That is: the *car* property name denotes *a single car*. We will here have an instance of the *cars* class for each relationship, as we would have in the relational approach (a tuple for each). The representation of our example becomes:

$$cars = \{[p \mapsto c_1], [p \mapsto c_2]\}$$

For example, starting with an empty relation, and connecting $p$ with $c_1$:

```
  cars.connect(p,c1);
```

leaves us with:

$$cars = \{[p \mapsto c_1]\}$$

Adding a new relationship is now easy: just add an entry to the set *cars* representing the relation.

## 4 Back to classes

Now, if we change associations this way, then what about classes? Remember that we liked to write the example as follows when embedding the relation as a property in class *Person*:

```
  class Person {
    var cars : {Car}
  }
```

We can keep this style, but now we have two different ways semantically to represent relations: the relational approach as just argued in the previous section, and then the data type view in the specification above. Do we want to support both views, and be able to write the above – as well as this (relational view):

```
  class Person {
    var cars : Car 0..*
  }
```

and should they mean the same thing: be given a relational interpretation?

## 5 Proposal

Based on the observations above, I suggest the following.

## 5.1 Introducing roles

I suggest that we keep the data type concept completely separate from the relation concept. I suggest that we introduce a new keyword for roles (**rol** or **role**) to represent relations:

```
'rol' roleName ':' className multiplicity ?
```

The type of a role identifier (the className) can only be a name (an identifier identifying a class) and the (optional) multiplicity characterizes the relation (and not the type as it has been up till now).

## 5.2 Associations

We can now write the relations as follows (here shown with a multiplicity):

```
assoc cars {
  rol person : Person;
  rol car : Car 0..*;
}
```

An instance of this class represents a single tuple in a relation: a relationship between (in this case) one person and one car. The above notation allows for $n$-ary relations (just more **rol** declarations).

## 5.3 Classes

When it comes to classes, I suggest that the example should be written as follows:

```
  class Person {
    rol cars : Car 0..*
  }
```

The meaning is that a relation is introduced between the set $Person$ and the set $Car$. That is, it is the same as writing an anonymous relation (no relation name):

```
  class Person {}

  assoc {
    rol person : Person;
    rol car : Car 0..*;
  }
```

Furthermore, this is **not** the same as writing:

```
  class Person {
    var cars : {Car}
  }
```

which in turn means that a *Person* object **contains** a (reference to a) set of cars. The effect is the same, however, which unfortunately gives us more than one way to model a concept such as a relationship. We can the discuss whether we need data types such as $\{Car\}$. However, I believe they are useful. The point perhaps is that if one uses the last class definition above, then it would not be visualized as a relation when thrown to MagicDraw. The property *cars* would just be perceived at the same level as a value attribute.

### 5.4 A bit on notation

I suggest that instead of the collection type construtors $\{T\}$, $[T]$ and $< T_1, T_2 >$, we write $Set[T]$, $List[T]$, and $Map[T_1, T_2]$ as in SCALA. It is just easier to read and one does not have to explain it to new-comers (to the same extent). So we would write the last class above as:

```
class Person {
  var cars : Set[Car]
}
```

## 6 Conclusion

We would like to keep the data type view since we are used to be able to talk about sets, lists and maps as data types. However, we are confronting a small dilemma. We also need to host the relational view within the same language, leading to some potential inconsistencies or multiple ways of writing the same thing. There is possibly a very quick fix to this problem. This is really the collision of formal specification and programing languages on the one hand and then the relational modeling world on the other.