An Introduction to K
A Language for Development of Systems

Klaus Havelund and Rahul Kumar

Jet Propulsion Laboratory
California Institute of Technology
California, USA

Abstract. ...

1 Introduction

2 Requirements

3 Modeling in SysML

4 Modeling in K with data types

Let us start with a very simple model. A spacecraft is a composition of hardware and software. To model this, we define three classes, named *Hardware*, *Software*, and *Spacecraft*:

```
class Hardware
class Software

class Spacecraft {
  var hardware : Hardware
  var software : Software
}
```

A class has a name and may contain a collection of definitions. In the case of the classes *Hardware* and *Software*, there are no such definitions, corresponding to making an abstraction: we say no more about these concepts at this point. A *Spacecraft* on the other hands consists of two so-called member declarations: a *hardware* field of type *Hardware*, and a *software* field of type *Software*. These are variables that can be assigned values of their respective types.

4.1 Refining the Hardware

Before we refine[1] the *Hardware* class, let's define a small collection of auxiliary concepts, such as a battery providing power, a power consuming unit, an instrument, etc.

[1] When we say we refine a class it means editing the original definition by adding more details. One can imagine support for a more formal approach where the old definition remains unchanged.

A battery has a maximal charge and a current charge, and can be recharged, as well as used, using the *update* function, which as argument takes the delta with which the battery should be updated (a negative number of power is used and a positive number if power is added).

```
class Battery {
  val maxCharge : Real
  var currentCharge : Real

  fun update(Δ: Real)
    pre currentCharge + Δ ∈ {0..maxCharge}
    post currentCharge ∈ {0..maxCharge}
  {
    currentCharge := currentCharge + delta;
  }
}
```

The *update* function is defined with a body, as well as a pre and post condition. The semantics of a pre-condition is that it must be true before the function is applied, otherwise the function result is ill-defined. The semantics of the post-condition is, that it is guaranteed to hold after the function has been applied.

A power consuming device is of the following type:

```
class PowerConsumer {
  var currentPower : Real
}
```

Here a variable represents the current power consumption. We can now define some power consuming devices, such as the radio and an instrument:

```
class Radio extends PowerConsumer

class Instrument extends PowerConsumer {
  name : String
  weight : Real
  power : Real
  operating : Bool

  fun toggle() {
    operating := !operating;
    if !operating then
      power := 0
    end
  }
}
```

An instrument has a name, a weight, a current power consumption, and a flag indicating whether it is turned on or not, which can be toggled with the *toggle* function.

Finally we can define the *Hardware* class:

```
class Hardware {
  type WheelNo = {| n : Int . 1 ≤ n and n ≤ 6 |}

  var wheels : Map[1..6,Wheel]
  var instruments : Set[Instrument]
  var battery : Battery

  val maxWeight : Real

  req instrumentsUnique :
    forall i1, i2 : instruments .
      i1 ≠ i2 ⇒ i1.name ≠ i2.name

  req notTooHeavy:
    instruments. collect (i → i.weight).sum() ≤ maxWeight;

  req controlPowerUsage :
    instruments. collect (i → i.currentPower).sum() ≤ battery.remaining ∗ 0.5
}
```

We first define a type *WheelNo* of wheel numbers, namely the numbers 1 to 6. *WheelNo* is a predicate subtype of the type *Int* of integers.

The class then contains four variables. The variable *wheels* is a finite mapping from wheel numbers to wheels. A mapping is essentially a function, but which is only defined on a finite domain, and which can be updated. The variable *instruments* denotes a set of instruments. The type `Set[Int]` is the type of all sets of integers. Each element of this type is a set of integers.

4.2 Refining the Software

The software in turn consists of flight software, which runs on the spacecraft, and ground software, which runs on grounding, sending commands to the spacecraft and receiving telemetry from the spacecraft:

```
class Softare {
   flight : FlightSoftware
  ground : GroundSoftware
}
```

All software shares common attributes, the *SoftwareBasis*:

```
class SoftwareBasis {
  type ModuleName = String
  type Language = String

  val usedLanguages : Set[Language] = {"Java","C","C++"}
```

```
  modules : Map[ModuleName,Module]

  codingStandards : Map[Language,List[Rule]]

  fun codingStandardCheck : Module → List[Int * Int]

  fun linesOfCode() : Int = modules.range().collect(m → m.code.length()).sum()

  req codeCorrect : forall module : modules.range() . standardChecker(module).isEmpty();
}
```

We first define the type of *ModuleName* module names, which are basically just strings. Such type definitions are just aliases. Hence in the scope of this definition the type *ModuleName* is equivalent to the type *Int*.

```
class Module {
  language : ProgrammingLanguage
  code : List[String]
}

enum ProgrammingLanguage{Java,C,Cpp}

class FlightSoftware extends SoftwareBasis {

}

class GroundSoftware extends SoftwareBasis {
  logs : List[Log]

  req logsSortedWrtTime :
    forall (log1,log2) : logs.pairs() . log1.startTime ≤ log2.startTime

  fun find(startTime: Int, endTime: Int): List[Event] =
    logs.select(log → log.startTime ∈ startTime .. endTime).flatten()
}

class Log {
  events : List[Event]

  req eventsSortedWrtTime :
   forall (e1,e2) : events.pairs() . e1.time ≤ e2.timex

  req eventsWellformed :
    events.forall(e → e.wellformed())
}
```

```
class LogEntry {
  fun wellformed: Unit → Bool
}
class Event extends LogEntry {
  kind : String
  fields : Map[String,String]
  fun wellformed() = schema(kind) subsetof fields
}
class Sampling extends LogEntry {
  sensor : String
  value : Real

  fun wellformed() = true
}
```

5 Modeling in K with relations

6 Reference manual for K

7 Conclusion