K: A Wide Spectrum Language for Modeling, Programming and Analysis*

Klaus Havelund, Rahul Kumar, Chris Delp and Bradley Clement
Jet Propulsion Laboratory, California Institute of Technology, Pasadena, California, U.S.A.

Keywords: Modeling, Programming, Constraints, Refinement, Verification, SMT, Analysis, SysML, Translation.

Abstract: The formal methods community has over the years proposed various formally founded specification languages based on predicate logic and set theory, typically with textual notations. At the same time the model-based engineering community has proposed often less formally founded languages such as UML and SysML, typically with graphical notations. Although the graphical notations have become highly popular in industry, we argue that textual notations can be attractive in many situations. We report on an effort to provide a textual notation for SysML, realized in a language named K. K supports classes, multiple inheritance, predicate logic and set theory. K contains programming constructs, and can thus be considered as a wide-spectrum modeling and programming language. We further explain the translation of a subset of this language to the input language of the SMT-LIB standard, and the application of Z3 for analysis of the generated SMT-LIB formulas. The entire effort is part of a larger effort to develop a general purpose SysML development framework for designing systems, in support of NASA's proposed 2022 mission to Jupiter's moon Europa.

1 INTRODUCTION

Modeling is the activity of formulating an abstract description of a system, for example to understand the system before it is implemented. Modeling includes activities such as requirements capture in the initial phases and design of higher-level architectural decisions in later stages. Modeling has been studied by various communities, of which at least two can be identified: the model-based engineering community and the formal methods community. The *model-based engineering* community has suggested modeling languages such as UML (OMG, 2015) and SysML (OMG, 2012), a variant of UML. These languages typically come equipped with graphical notations (concrete syntax). Both UML and SysML have been designed by the OMG (Object Management Group) technology standards consortium. SysML is meant for systems development more broadly considered, including physical systems as well as software systems, in contrast to UML, which is mainly meant for software development. The graphical notations have received a high degree of popularity in industry due to their two dimensional format, also sometimes referred to popularly as *boxology*: boxes and arrows. However, drawbacks of these languages include lack of precise semantics, lack of analysis capabilities, tedious GUI operations in tools supporting the graphical notations, requiring lots of visual real estate even for simple models, as well as large volumes of technologies. Learning UML and SysML is not just learning very large languages, it is also learning a large set of additional tools needed to work with models. We formulate the hypothesis that some of these drawbacks in part are due to the lack of a simple textual notation, at a size comparable to a programming language notation. As evidence of this hypothesis, one may observe, that software developers mostly prefer to program in textual languages.

From an even earlier point in time, since the 1960s, the *formal methods* community, part of the computer science community, and closely connected to the programming language community, has proposed numerous formally founded specification languages with textual notations. Several of these are based on predicate logic and set theory. These languages are, compared to UML and SysML, concise, small, well defined in the form of semantics, and in recent time well supported with analysis capabilities. The obvious observation is that it might be fruitful to study the interaction between the two classes of

*The research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

Havelund, K., Kumar, R., Delp, C. and Clement, B.
K: A Wide Spectrum Language for Modeling, Programming and Analysis.
DOI: 10.5220/0005741401110122
In *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2016)*, pages 111-122
ISBN: 978-989-758-168-7

languages. Consider furthermore that programming languages are gaining in abstraction, such as for example combining object-oriented and functional programming. An example is Scala, which has many commonalities with very early formal specification languages, such as for example VDM (Bjørner and Jones, 1978), and specifically its object-oriented variant VDM^{++} (Fitzgerald et al., 2005). The study should therefore include the interaction between languages with graphical notations, and languages with textual notations, such as formal methods modeling languages, and programming languages

We report on an effort to develop a textual notation for SysML. For this purpose we have developed a language named K (Kernel language) with a textual notation inspired by notations used in the formal methods community. K corresponds to SysML's class diagrams plus constraints. It is our plan to map other SysML language concepts into this kernel, rather than extending the K language to incorporate the rest of SysML. K supports object-oriented concepts such as classes, multiple inheritance, and object instances. The contents of classes can be typed values, including functions, and constraints over these expressed in higher-order predicate logic. K also contains programming constructs such as variables/properties, assignment statements, and looping constructs, and can as such be seen as a wide-spectrum modeling and programming language. The K language provides an alternative to modeling with the mouse in tools that typically support SysML: namely writing textual models in the K notation directly, just like one normally writes programs. K can furthermore be seen as a vehicle for giving semantics to SysML, providing analysis capabilities. It is our hypothesis, that modeling can be seen as programming in a language where some parts of the model (program) at any point in time are executable, and some maybe are not (yet). Supporting this hypothesis is the large number of language constructs shared by modeling and programming languages.

The idea of merging modeling and programming in one language has been suggested before, as will be discussed in the related work section. Although K is not much different from previously suggested formalisms, our contribution is the creation of K specifically in support of a SysML model engineering tool set under development, to be used by designers of the proposed 2022 mission to Jupiter's moon Europa, also referred to as the Europa Clipper mission concept (Europa Clipper Mission, 2015). The resulting tool set will support graphical SysML modeling using MagicDraw (MagicDraw, 2015), as well as browser-based model viewing and editing, including of textual K models. A first-order subset of K is furthermore translated to the input language of the SMT-LIB standard, and currently processed with the Z3 SMT solver for proving satisfiability of class definitions (are the constraints consistent?), and model finding (find variable assignments satisfying constraints, for example used in task scheduling). The contribution here is the handling of (multiple) class inheritance, which is typically not supported by similar languages translated to SMT-LIB, as well as the allowance of recursive class definitions. Multiple inheritance is a crucial part of SysML, and therefore of K.

The paper is organized as follows. Section 2 introduces a subset of the K language through an example, which is similar to examples typically used to illustrate such formal specification languages. Section 3 outlines the translation from K to the SMT-LIB input language for the purpose of analysis of K models. This section is based on a different example illustrating how K is actually currently used at JPL. Section 4 explains the integration of K within the SysML development framework, as well as the usage of this. Section 5 discusses related work, and finally Section 6 concludes the paper.

2 INTRODUCTION TO K

In this section we introduce the K language. We use the K model in Figure 1 as our running example for discussing core concepts in K. The example shows a model of a file system modeled using K. It is intended to be a basis for discussing language features, and not a complete model of a file system.

K is a high level textual language which supports multiple paradigms. It allows one to create *packages*, which are collections of *classes*. Packages can be *imported* by other K files. Line 1 in Figure 1 shows an example of a package declaration. Classes, as in other object-oriented languages, provide a means for abstracting and grouping properties (variables). In K, classes may contain properties, functions (there is no distinction between functions and methods), and constraints (requirements). Scoping rules in K are similar to languages such as Java and C++. Lines 9 – 12 in Figure 1 declare an Entry class, which contains two members: property name of type String, and function size that takes no arguments and returns an Int. The function implementation is not specified for function size. String is one of the six primitive types provided by K: Int, Real, String, Char, Unit, and Boolean. K also provides the following collections:

Bag: collection of items not subject to any order or uniqueness constraints.

```
1  package examples.fs
2
3  BLOCK_SIZE : Int
4
5  type Byte =
6    {| b : Int :- 0 <= b &&
7       b < 256 |}
8
9  class Entry {
10   name : String
11   fun size : Int
12 }
13
14 class Dir extends Entry {
15   var contents : Set[Entry]
16   fun size: Int {
17     contents.collect(e →
18       e.size()).sum()
19   }
20 }
21
22 class File extends Entry {
23   contents : Seq[Block]
24   req size() <=
25     contents.size() *
26       BLOCK_SIZE
27 }
28
29 class Block {
30   bytes : Seq[Byte]
31   fun size: Int { bytes.size() }
32   req size() <= BLOCK_SIZE
33 }
34
35 class FS {
36   dir : Dir
37
38   fun mkDir(n: String) : FS
39     pre !exists
40       e : dir.contents :-
41           e.name = n
42     post $result.dir.size()
43            = dir.size()
44   {
45     newDir : Dir = Dir(name::n)
46     nc : Set[Entry] =
47       dir.contents + newDir
48     FS(dir::
49       Dir(name::dir.name,
50         contents::nc))
51   }
52
53   fun rmDir(n: String) : FS
54     pre exists e : dir.contents :-
55           e.name = n
56     post $result.dir.size()
57            <= dir.size()
58 }
```

Figure 1: A simple model of a file system using K.

Seq: collection of items subject to an ordering, but no uniqueness constraints.

Set: collection of items subject to uniqueness, but no ordering constraints.

OSet: collection of items subject to uniqueness, as well as ordering constraints.

K provides *predicate subtypes*. Line 5 specifies a subtype named Byte, which is derived from the Int type but restricted to values between 0 and 256. K allows classes to inherit from one or more classes. For example, class Dir, specified on lines 14 – 20 extends the Entry class. As with other languages, inheritance causes the child classes to inherit the instance variables and functions of the parent classes, but in addition, in K, the child classes also inherit the constraints from the parent classes. In the case of multiple inheritance, K requires that the property names be unique across all classes. Functions on the other hand may be overloaded by changing the function signature. Both class File and Dir inherit from class Entry. Line 15 declares the variable contents using the keyword var, indicating that this variable is mutable (can change value). Variables introduced without this keyword (or with the keyword val) are constants. Lines 16 – 19 in Figure 1 show the implementation for the size function in the Dir class. It makes use of the sum function, that is provided by K for all numerical collections. The size function is the same as declared in class Entry. Currently, function bodies cannot be declared more than once along an inheritance path. Functions may take an arbitrary number of arguments and return a single value. K also provides tuples to group objects together. On line 32, we see a constraint being specified for class Block using the req (require) keyword. The constraint specifies that the size function of Block should always return a value that is less than or equal to the value specified in the global property SIZE_OF_BLOCK (left unspecified). Any number of constraints can be specified at the global scope or within classes. Constraints are Boolean expressions, that restrict the values variables can take. Constraints in a class can be considered class invariants.

Class FS (for FileSystem) contains two functions: mkDir and rmDir. The mkDir function takes a single argument (n of type String) and returns a FS object which contains one additional directory entry that has name n. The rmDir function has no body specified. Both functions are defined along with a *function specification*. Function specifications are a list of *pre* and *post* conditions that describe the precondition and postcondition of the function. Any number of specifications may be provided. Line 39 specifies the precondition for function mkDir with the use of an *existential* quantifier. It specifies that when creating a directory in the file system, the given name n should not exist in the current set of entries in the file system. K provides both *existential* and *universal* quantification in its expression language. For the same function, line 42 specifies the postcondition. $result is a reserved word that refers to the return value of the function. It can only be used when specifying postconditions. The postcondition for mkDir specifies that function mkDir returns a FS object that has the same size as the current FS object, which was used to create the new directory. Lines 44 – 51, the body of function mkDir, form a block consisting of the declaration of two constants: newDir and nc, followed, in line 48, by the creation (and return) of a new FS object by calling the constructor for class FS. Note that the entries of a block are not separated by semicolon (';'). In fact, K does not have a semicolon (nor newline) as statement separator, as for example seen in the programming language Scala. The only argument provided to the FS constructor is a Dir object which contains one additional Dir entry whose name is n. K provides constructors automatically for all classes where the arguments are *named arguments*. Each named argument is of the form 'member :: value' where the '::' notation is used as a form of assignment. Multiple named arguments can be provided as a comma delimited list. It is not necessary to specify a value for all members of a class. Any members that are specified in a constructor call are assigned the specified value, and the rest are left underspecified. Function rmDir is specified with no body, but only function specifications. The function specifications require that function rmDir only execute if the provided directory n exists in the current object's contents. The postcondition specifies that the resulting FS object should be either the same size or smaller relative to the current object.

Expressions in K are the core of the language. Expressions in K allow one to write assignment statements (side-effects), binary expressions (such as and, or, implication, iff etc.), logical negation, arithmetic negation, quantification, 'is' for checking type, and 'as' for type casting. Any expression can make use of other defined constructs such as variables, function application, lambda functions, and dot expressions. K also supports control expressions such as if-then-else, while, match, for, continue, break, and return. These expressions are similar to control expressions provided in programming languages such as Scala or Java. A detailed description of the expression language is beyond the scope of this paper.

K also provides *multiplicities* as part of the language. Multiplicities in K are influenced by similar concepts in languages such as UML/SysML. In K, multiplicities can be used as a short hand for specifying collections and also restricting the size of collections. Figure 2 shows a K model of a Person that has various member properties, and the corresponding inferred type for each member property. We will analyze each of these individually.

Each Person can have exactly one mother. This is specified by line 4. No explicit multiplicity is specified, which makes it a singleton. A Person can also have many unique children, which is specified by line 5 using the Set collection. Line 6 specifies that a Person may have many cars. It is written using a modifier and a multiplicity, which semantically translates to a Set (K default for a multiplicity is Bag) of Car. Finally, a person may own one or more portfolios (prtflios, specified to have a multiplicity of 1 or more), where each entry itself is a Set of Stock. This translates to prtflios being a Bag of Set[Stock] with at least 1 entry and no upper limit.

SysML models can also carry meta data information in them (sometimes introduced by tools). To accommodate for this, K also provides the *annotation* construct. New annotations can be created and applied to classes, expressions, functions etc. Currently, each annotation has a name and a type.

K also provides single line comments using '--' at the beginning of the line, and block comments using '===(=*)' as the token for the beginning and the end of the comment.

2.1 K Type Checking

The K type checker performs basic checks on the provided input to ensure naming and type consistency. It is used to ensure that all declarations, expressions, annotations etc. are logically sound and reference names (functions, members, variables) that exist and are type consistent in the given context. Type information for all expressions and any other inferences made by the type checker are saved and made available to all other analyses/modules in the K tool chain. Further, the type checker imposes a stricter set of rules on the provided input to ensure that it can be completely and correctly translated to SMT. More details are provided in Section 3. The type checker is implemented as a stand alone module, which is invoked after the AST has been constructed by a visitor (interfacing with ANTLR). The implementation is done using Scala.

3 TRANSLATING K TO SMT-LIB

In this section we illustrate the translation from K to the SMT- LIB input language. SMT-LIB (SMT-LIB,

```
class Stock
class Car
class Person {
  mother : Person                        Person
  children : Set[Person]                 Set[Person]
  unique cars : Car [*]                  Set[Car]
  prtflios : Set[Stock] [1,*]            Bag[Set[Stock]]
}                                          req prtflios.size() >= 1
```

Figure 2: Example model (left) and inferred types (right) for members of class Person.

2015) is the standard "satisfiability modulo theories library" for SMT solvers. The standard is used by numerous SMT solvers, allowing comparison between systems (for example in competitions). In addition, it allows systems generating SMT-LIB formulas to target any SMT solver processing this standard. In our case we use the Z3 SMT solver (De Moura and Bjørner, 2008) to process the generated formulas, but anticipate targeting other solvers in the near term.

3.1 The Source K Model

The translator currently covers a first-order logic subset of the K language, corresponding to the model of a SpaceCraft shown in Figure 3. The translated subset includes classes, multiple inheritance, properties of primitive types (Bool, Int, and Real), user-defined class types, tuple types (cartesian product), functions, pre/postconditions, and constraints. Functions, pre/-postconditions, and constraints can be specified in a rich expression language supporting conditionals, class constructors, dot notation for accessing properties in objects, and universal and existential quantification. Sets are under development but not covered here. Currently not translated constructs include type parameterized classes, statements with side-effects (assignment) and their corresponding looping constructs, functions as first class citizens, type abbreviations and predicate subtypes, as well as multiplicities, which will be treated as collection types. Recursive functions can currently not be defined using function definitions, but can be specified by providing function signatures plus separate constraints.

The example illustrates the features of K that have been used by engineers at JPL until the time of writing. The emphasis of these models is on *structure* of artifacts and *scheduling* of events. The class Thing is meant to represent entities that have weight. Instruments, its radio sub-classes, and the SpaceCraft class itself, inherit from class Thing. Class Instr defines a power level. Requirements in the form of Boolean constraints are imposed on power and weight. The SpaceCraft class makes instances of instruments, defines a combined sum instrWeight, and a constraint on it with additional requirements. Such elements of a model are so-called *structural* elements, what one would normally see in a SysML class diagram.

The spacecraft in addition contains a system manager, representing the software on board. For the purpose of illustration, the system manager is defined as a small *scheduler* of three events: a bootUp event, re-booting the flight software computer, an initMem event, initializing the computer memory, and a tkPic event, taking a picture. An event is an instance of the Event class, which defines an event as having a start time and an end time appearing after the start time. In addition, the Event class defines a function after, which as argument takes another event 'e', and returns true if the event (this) occurs after 'e'. The definition of the after function is inspired by Allen logic (Allen, 1984). Finally, the model contains an instance ShRaan of type SpaceCraft.

Given the spacecraft model, the general proof-theoretic problem we want an answer to is whether our classes are logically consistent. That is, whether the constraints of each class are consistent (do not evaluate to *false* such as for example is the case with: '$x < 0 \wedge x > 0$'). From a semantics point of view, it means that for each class there exists at least one instance (object) of that class that satisfies the constraints. An SMT solver demonstrates satisfiability by finding a model satisfying the constraints (model finding). This model furthermore represents a schedule of the events in the system manager.

3.2 The Translation to SMT-LIB

The SMT-LIB language (from here on referred to as SMT-LIB) is equipped with a textual notation, and supports typed first order predicate logic plus various theories, including for example arithmetic, uninterpreted functions, and arrays. The syntax is LISP-like, meaning for example that function calls such as $f(42, false)$ have the form $(f\ 42\ false)$. For the 51 line K model in Figure 3, the translator generates 333 lines of uncommented SMT-LIB code (additional comments are generated to make the output easier for humans to read). We shall below show formulas from each category of formulas generated, covering all the categories. Our main challenge in translating K to

```
class Thing {weight : Int}

class Event {
  start : Real
  end : Real
  req start >= 0.0 &&
      end > start
  fun after(e : Event)
        : Bool {
    start >= e.end
  }
}

class Instr
      extends Thing {
  power : Real
  req power >= 0.0
  req weight > 0 &&
      weight <= 1000
}

class SmplRadio
      extends Instr
class SmrtRadio
      extends SmplRadio

class SysMngr {
  bootUp : Event
  initMem : Event
  tkPic : Event
  req tkPic.after(initMem)
      &&
      tkPic.after(bootUp)
}

class SpaceCraft
      extends Thing {
  instrWeight : Real
  radio : Instr
  camera : Instr
  software : SysMngr

  req instrWeight =
        radio.weight +
        camera.weight

  req notTooHeavy:
        instrWeight <= 999
}

ShRaan : SpaceCraft
```

Figure 3: A simple K model of a spacecraft.

SMT-LIB is how to translate classes supporting (multiple) *inheritance* and *recursive* references between classes. This will be illustrated in the following.

Classes, Objects, and the Heap. Let's first translate a simple class, such as class Thing. We have chosen to translate classes to the SMT-LIB concept of *datatypes*. A datatype in SMT-LIB corresponds to the classical notion of an algebraic datatype: a named record, with a constructor function that when applied to a sequence of values generates a value of the datatype, while the values can be retrieved using selector functions. The class Thing can be represented in SMT-LIB as follows.

```
(declare-datatypes () ((Thing
  (mk-Thing (weight Int)))))
```

This declaration declares the datatype Thing, the constructor mk-Thing, which can be called on a value w of type Int as follows: (mk-Thing w), to produce a value in type Thing. Reversely, given a value o in type Thing, we can retrieve the weight by applying the selector function weight to o as follows: (weight o).

Consider now the following schematic example of two mutually recursive classes, a situation often occurring in SysML modeling (relationships between two classes) as well as in programming (i.e. linked lists). Note that due to the declarative nature of K, it is possible to initialize objects of such classes in a recursive manner even without programming with side-effects. If no constructors are applied, the solver will assign objects.

```
class A {       class B {
  b : B           a : A
}               }
```

The following translation of this model to the SMT-LIB datatypes A and B is **not** well-founded since it contains recursion between A and B (it is illegal SMT-LIB).

```
(declare-datatypes () (
  (A (mk-A (b B)))
  (B (mk-B (a A)))))
```

The solution is to operate with *references* to objects rather than objects directly, exactly as done in any runtime system for an object-oriented programming language. In other words, we need a *heap* mapping references to objects. For this purpose we define the type of references as integers.

```
(define-sort Ref () Int)
```

We can now in SMT-LIB use Ref as the type of properties whose type in K is a class, they will now denote references to objects of the class. This is illustrated by the following definition of the SpaceCraft datatype.

```
(declare-datatypes () ((SpaceCraft
  (mk-SpaceCraft
      (weight Int)
      (instrWeight Real)
      (radio Ref) (camera Ref)
      (software Ref)))))
```

Observe how the fact that SpaceCraft inherits from Thing is modeled by the inclusion of the weight field from Thing. Inheritance is simply modeled by property inclusion in this manner. In order to define a heap, we need a datatype that represents all the objects that can possibly be stored in the heap. The following datatype Any represents all the datatypes for the individual classes, by lifting them to this single type (null is a zero argument constructor). The type Any corresponds to Java's type Object.

```
(declare-datatypes () ((Any
  (lift-Thing
    (sel-Thing Thing))
  (lift-Instr
    (sel-Instr Instr))
  (lift-SpaceCraft
    (sel-SpaceCraft SpaceCraft))
  ...
  null)))
```

Now we can define the heap as an array from references of type Ref to Any.

```
(declare-const heap (Array Ref Any))
```

Accessing the Heap. We first define a function deref, which when applied to a reference returns the Any object at that entry.

```
(define-fun deref ((ref Ref)) Any
  (select heap ref))
```

With this function we are now ready to define functions, which can test what kind of object is at a certain location in the heap, as well as retrieve that object. The following functions perform these two tasks for the case of the Instr objects (for each datatype constructor C, SMT-LIB generates an is-C function that can determine whether an object is constructed with the constructor).

```
(define-fun deref-is-Instr
  ((this Ref)) Bool
  (is-lift-Instr (deref this)))

(define-fun deref-Instr
  ((this Ref)) Instr
  (sel-Instr (deref this)))
```

As we have seen, K classes can contain properties of types that are classes. For example the SpaceCraft class contains a property radio of type Instr. In an object-oriented language like K with inheritance, such a property can denote any object that is of type that either is equal to, or sub-classes Instr. In order to formulate invariants on objects of class SpaceCraft, we therefore need to be able to determine whether a radio object is equal to, or sub-classes Instr. This task is performed by the following function, the body of which is a disjunction between the three alternatives.

```
(define-fun deref-isa-Instr
  ((this Ref)) Bool
  (or
    (deref-is-Instr this)
    (deref-is-SmplRadio this)
    (deref-is-SmrtRadio this)))
```

Getters of Properties in Classes. Functions and requirements access properties. An example is the expression weight > 0 in class Instr. These accesses are wrapped into *getter* functions. As an example, the weight property of the class Instr can be accessed with a call of the following function, named Instr!weight (SMT-LIB allows symbols such as '!' in names, to be discussed further below), on a reference that is assumed to refer to an Instr object.

```
(define-fun Instr!weight
  ((this Ref)) Int
  (weight (deref-Instr this)))
```

The above definition assumes that the this reference denotes an Instr object, and not an object of any subclass on Instr, hence the '!' symbol (for *exact! class*) in the name. This is sufficient when checking satisfiability of the class Instr class itself. However, when checking the satisfiability of, for example, the SpaceCraft class, which *contains* a property of type Instr, as for example radio : Instr, we have to assume that radio in addition potentially can refer to any object of a class that sub-classes Instr, which in this case is either SmplRadio or SmrtRadio. This is achieved with the following alternative getter function, named Instr.weight, for the weight property of the class Instr.

```
(define-fun Instr.weight
  ((this Ref)) Int
  ; if
  (ite (deref-is-Instr this)
    ; then
    (weight (deref-Instr this))
    ; else if
    (ite (deref-is-SmplRadio this)
      ; then
      (weight (deref-SmplRadio this))
      ; else
      (weight (deref-SmrtRadio this))
    )))
```

Each line in the body is preceded with a comment using the comment symbol ';', explaining the structure of the LISP version of '**if** e_1 **then** e_2 **else** e_3', which is '(**ite** e_1 e_2 e_3)'. The reason for not just using the latter more general function Instrument.weight for all accesses to the weight property is that conditionals make it harder for an SMT solver. Even moderately sized expressions with several accesses to variables become unsolvable in reasonable time in the presence of such conditional expressions.

Functions. Functions are translated directly to SMT-LIB functions. Each function is translated in two versions, corresponding to the two versions of the getter functions, and named using respectively className!functionName and className.functionName, to suggest which getter functions are called inside the function, again depending on the calling context (whether this refers to the exact class or potentially a sub-class). As an example, the following is the translation of the after function in the class Event, only showing one of the two versions, which are the same in this case.

```
(define-fun Event.after
  ((this Ref)(e Ref)) Bool
  (>= (Event.start this)
  (Event.end e)))
```

The first parameter is a reference (named this) of type Ref. The this reference is meant to refer to the object upon which the function is called. Consider for example a call like: tkPic.after(initMem) in line 31 of Figure 3. Here tkPic denotes a reference to which the parameter this is bound. The second parameter is the user-provided parameter.

Invariants and Assertions. We are finally able to present how class invariants are generated and asserted. These validate the satisfiability of our classes. The invariant for a class is generated as a function that as argument takes a this reference to an object of that class. Let's take the example of the SysMngr class. The generated invariant is the following.

```
(define-fun SysMngr.inv
  ((this Ref)) Bool
  (and
    (deref-isa-Event
      (SysMngr!bootUp this))
    (deref-isa-Event
      (SysMngr!initMem this))
    (deref-isa-Event
      (SysMngr!tkPic this))
    (and
      (Event.after
        (SysMngr!tkPic this)
        (SysMngr!initMem this))
      (Event.after
        (SysMngr!tkPic this)
        (SysMngr!bootUp this)))))
```

The body of this function is a conjunction of the conditions that have to hold on the SysMngr object referred to by this. There are four such: three for the property definitions in lines 28 – 30 in Figure 3, and one for the requirement on line 31. Each of the property definitions results in a condition that verifies that the property is of the right type, in these three cases: that each of the properties bootUp, initMem, and tkPic, are objects of any sub-class of class Event (the use of 'isa'), although in this case there are no sub-classes of Event. The last condition, corresponding to the requirement, illustrates how functions are called, in the case the function after.

We are now finally ready to assert the well-formedness of the model. For each class two assertions are generated, one that asserts the existence of an object of the class in the heap, and one asserting that every object of that class in the heap satisfies the invariant of that class. Below are these two assertions for the SysMngr class.

```
(assert (exists ((this Ref))
  (deref-is-SysMngr this)))

(assert (forall ((this Ref))
  (=>
    (deref-is-SysMngr this)
    (SysMngr.inv this))))
```

Solving the Model. Given the generated SMT-LIB model outlined above, an SMT solver following the SMT-LIB standard can determine whether the model is satisfiable. Our currently used SMT solver is Z3. If the model is **not** satisfiable, the solver will just return 'not satisfied'. One can in this case analyze subsets of the model, eliminating assertions to discover which assertions caused the model to become unsatisfiable, in the best case the minimal set of such assertions. We are working on such a violation explanation capability.

If the model on the other hand is satisfiable, an assignment to variables in the model will be returned by the solver. In our case the model outlined above is satisfiable and solves in 2 seconds. The returned assignment is shown in Figure 4. This view has been produced by processing the output from Z3, which is less comprehensible. The assignment shows the following. The outermost ShRaan property in the heap denotes a SpaceCraft object. This object contains various fields, for example the weight property with the value 18, and the software property, which denotes the reference (of type Ref) 21. This reference in turn denotes a SysMngr object containing three references bootUp (25), initMem (26), and tkPic (27), each of which are events. Due to the constraint in line 31 of Figure 3 these events have been *scheduled* such that the taking of the picture occurs after the boot as well as after the memory initialization. This can be seen from the fact that the end times of the boot and memory initialization events at references 25 and 26 are less than the start time of the take picture event at reference 27. Note that the values suggested by the SMT solver are not necessarily realistic, although they satisfy the provided constraints.

4 K IN PRACTICE

Currently, K is used to analyze models created for the NASA Europa Clipper Mission Concept. Figure 5 gives an overview of the usage scenarios for the K language and tool chain.

```
+--------+-----------------------------------------------------------------+
|Variable|Value                                                            |
+--------+-----------------------------------------------------------------+
|ShRaan  |SpaceCraft(weight::18, instrWeight::966.0,                       |
|        |            radio:: Ref 19, camera:: Ref 20, software:: Ref 21)|
|Ref 19  |SmplRadio(weight::360, power::2331.0)                            |
|Ref 20  |Instr(weight::606, power::4770.0)                                |
|Ref 21  |SysMngr(bootUp:: Ref 25, initMem:: Ref 26, tkPic:: Ref 27)       |
|Ref 25  |Event(start::9659.9088, end::9660.9088)                          |
|Ref 26  |Event(start::7854.0, end::7855.0)                                |
|Ref 27  |Event(start::17257.0, end::21593.0)                              |
+--------+-----------------------------------------------------------------+
```

Figure 4: Output of the K tool chain for the spacecraft example.

The typical scenario involves modelers (of the Europa Clipper mission concept) creating SysML diagrams in a tool such as MagicDraw and saving them to a central model repository. This database of models is accessible via a REST API. The input to the REST API is a unique identifier for a node (typically a SysML package) in the model, and the result is a list of all the nodes that are part of the package specified in the input. The result is provided as an array of JSON objects, where each object contains information such as name, type, owner, etc. Typically, the types of objects are classes, constraints, expressions, and member properties. The K tool chain takes this input and converts each node in the list of nodes to a corresponding K AST object. Since the list of nodes received from the REST API is unordered and unstructured, we perform multiple passes on the list of nodes. The first pass is performed to create the list of classes in the model, followed by passes to populate properties and constraints in each class. Once the K model has been constructed, the K tool chain proceeds normally with type checking and SMT analysis. Currently this scenario is based on a *pull* methodology where a modeler has to initiate the K based translation and analysis. In the future, we plan on automating this effort and have it be executed on a regular cadence with results made available through the model database to a web application.

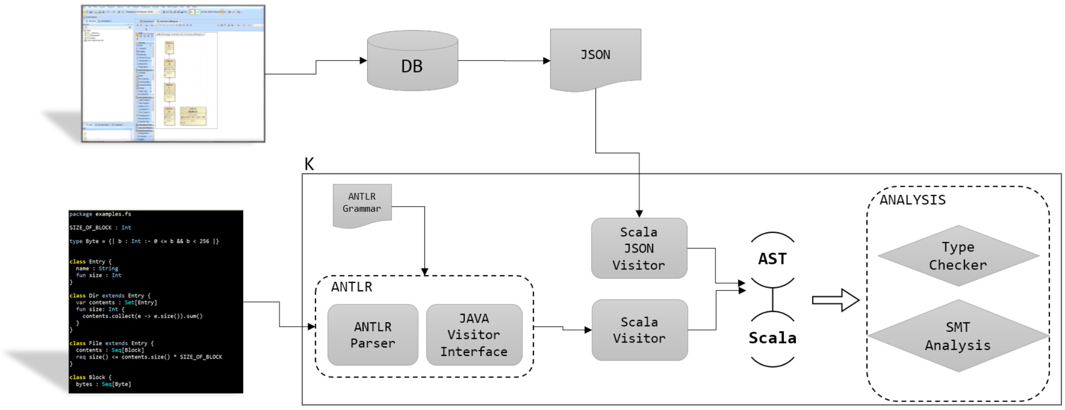
A second common scenario for using K is via the web browser (K, 2015). We have created a simple HTML based K code editor along with the functionality to invoke the K tool chain from the web browser. This page is used for purposes of teaching, learning, exploring, and prototyping with K. The web page also provides a tutorial, documentation, and examples.

Finally, the K tool chain is also available as a binary download for all major operating systems. Users may download the binaries and invoke the tool chain from the command line. We expect that certain models of Europa Clipper mission concept are created and analyzed directly as K models. Currently, there are two such salient examples where K was used to create and analyze requirements. The first model is a series of scheduling constraints, which after modeling in K, were successfully analyzed in less than 20 seconds. The results of the analysis also discovered a scheduling problem, that was later successfully confirmed by a significantly more cumbersome manual analysis. The second model contains a series of high level constraints that are analyzed using K for satisfiability. This model solves in around 30 seconds. Due to JPL's information release restrictions, such details cannot be shared at this time.

5 RELATED WORK

K is a wide spectrum language with a textual notation, containing concepts such as classes, inheritance, properties, functions, and expressions. Expressions in K are very rich and provide a basis for expressing any higher order logic formula succinctly. The language also contains constructs such as lists, sets, loops etc. for performing general sequential programming, functional programming, and object oriented programming. The main goal is to automatically translate SysML diagrams and models to K and perform various types of analysis such as type checking, satisfiability checking, and potentially execution.

The formal methods community has studied and investigated wide spectrum specification languages in detail. (Bjørner and Jones, 1978; Bjørner and Jones, 1982; Jones, 1990; Jones and Shaw, 1990) present the VDM language, which provides a combination of procedural and functional programming along with sets, lists, maps, and higher order predicate logic in proper mathematical notation. A natural evolution of VDM was to introduce object orientation, as shown in VDM^{++} (Fitzgerald et al., 2005). The RAISE specification language (George et al., 1992) is another wide spectrum language that takes inspiration from VDM, Z (Spivey, 1988) and algebraic specification languages. More recently, Asml presented in (Gurevich et al., 2005) presents a language where all operations operate on algebras. Alloy (Jackson, 2012) is



```
package examples.fs

SIZE_OF_BLOCK : Int

type Byte = {| b : Int :- 0 <= b && b < 256 |}


class Entry {
  name : String
  fun size : Int
}

class Dir extends Entry {
  var contents : Set[Entry]
  fun size: Int {
    contents.collect(e -> e.size()).sum()
  }
}

class File extends Entry {
  contents : Seq[Block]
  req size() <= contents.size() * SIZE_OF_BLOCK
}

class Block {
  bytes : Seq[Byte]
```

Figure 5: K in practice.

a language intended for describing structure and exploring the design of the structure through the specification of constraints. Alloy also provides seamless integration with a SAT solver. FORMULA (Jackson et al., 2009) also presents a language for logic programming where the model is then analyzed by SMT solvers. FORMULA focuses on requirements for incremental refinement purposes and design exploration.

In contrast to specification languages, high level programming languages have also been developed over time. SML (Standard ML) (Milner et al., 1997), Ocaml (OCaml, 2015), and Haskell (Jones, 2003) belong to the same family of functional programming languages with some support for classes and object orientation. Python (Python, 2015) also provides a unique perspective on combining object oriented and functional programming. In our view, Scala (Scala, 2015) does so to the fullest extent. The close relationship between Scala and VDM is discussed in (Havelund, 2011). Fortress (Fortress, 2015) introduced built-in notation for sets, lists, and maps, very much resembling the notation in VDM.

Yet another class of recent languages has investigated introducing specification constructs in programming languages in the form of design-by-contract (pre/postconditions and class invariants). Examples of such languages include Eiffel (Meyer, 1988) and SpeC$^{\#}$ (Barnett et al., 2011). Scala also attempts at providing such constructs, but through the use of library functions on functional programs (Odersky, 2010). Finally, The JML language (Leavens et al., 1998) allows to write design-by-contract specifications for Java as comments. These specifications and constructs are ignored by the standard Java compiler. Rather, they are processed with special tools; thus, in some cases making them less valuable to the language.

With the great improvements being made to SAT and SMT solvers (SMT-LIB, 2015; De Moura and Bjørner, 2008), programming languages now are also built with verification as a primary goal. Dafny (Leino, 2010) provides explicit support for specifications in the program that can be used to specify functional correctness constraints for programs. These constraints and specifications are verified using the Boogie (Barnett et al., 2006) verifier that uses Z3 as the underlying SMT solver. Dafny is an excellent language for performing verification, but lacks support for inheritance and class invariants, which are a necessity for the kind of models we deal with. Similarly, SpeC$^{\#}$ also provides support for verifying the user provided specifications. Why3 (Bobot et al., 2011) provides a rich language for specification and programming, called WhyML. Why3 relies on external theorem provers (automatic and interactive) to then verify the specification. Additionally, using WhyML, one can also generate Ocaml programs using a correct by construction automatic extraction procedure. In contrast to using automated provers, interactive theorem provers such as PVS (Owre et al., 1992; PVS, 2015), Coq (Barras et al., 1997), and Isabelle (Nipkow et al., 2002), also provide languages to create specifications and provide constraints, which can then be discharged via a user guided process. Such tools allow for proving much more complex properties, but tend to be laborious and non-trivial to use.

Past research in formal modeling has also resulted in various attempts at combining formal and semi-formal languages. Work in (Lausdahl et al., 2009) presents an approach to translate between UML and VDM^{++}, and (Kim et al., 2005) presents work on integrating a formal language such as Object-Z with UML in a single combined framework. To the best of our knowledge, these approaches have not focused on translating constraints and integrating with an SMT solver, and have been primarily focused on UML (not SysML).

K is very close in spirit and ideology to many of the aforementioned languages, but differs in many respects as well. For example, due to the application environment of K being targeted to SysML models, proving class satisfiability and model finding are of prime importance, something for which K is optimized. K also provides support for specifying and proving specifications with multiple inheritance, something SysML models depend on greatly. Integration with SMT solvers has proven to be useful not only for proving class satisfiability, but also for model finding (including scheduling) and *model exploration*. In model exploration, users manually explore the range of satisfiable solutions for the given model using iterative refinement techniques (changing constraints manually).

6 CONCLUSIONS

We have presented an overview of the K language in this paper. K is intended to be used in a modeling environment for proving satisfiability of SysML models and exploring solutions to various types of models, such as structure, planning/scheduling, etc. We have also presented in detail, our methodology for performing automatic translation of K models to SMT-LIB, and using an SMT solver such as Z3 to perform semantic model finding. Using manual methods of creating K models from SysML models and reference materials, we have already observed K provide value in the modeling environment by discovering unsatisfiability of scheduling problems in the proposed Europa Clipper mission concept, which was confirmed by external manual analysis. In our current experience, K seems to be sufficient for creating small to medium sized SysML models and proving properties about them. Concerning problems faced, a main challenge of course is the higher-order nature of K, requested by mission engineers (expressiveness prioritized over guaranteed analyzability). SMT-LIB is generally first-order. Some problems are a consequence of using SMT-LIB solvers, which struggle with the combination of arrays (used for the heap and for sets) and universal quantification. Additionally, the use of Real numbers and arithmetic on them is also a known SMT challenge, especially in the context of arrays. We are now in the process of creating tools to automatically translate SysML models to K models (and back) and perform analysis on them using the K infrastructure. This will make it possible to view a model as graphics as well as in text. The translation of K needs to be extended to cover more constructs, including statements with side-effects. Other challenges include making K executable, for example by translation to Scala, including executing OCL-like expressions; providing support for reflection such that models can query themselves; and making the K language and textual notation user-extensible.

REFERENCES

Allen, J. F. (1984). Towards a general theory of action and time. *Artificial Intelligence*, 23:123–154.

Barnett, M., Chang, B.-Y. E., DeLine, R., Jacobs, B., and Leino, K. R. M. (2006). Boogie: A modular reusable verifier for object-oriented programs. In *Formal methods for Components and Objects*, pages 364–387. Springer.

Barnett, M., Fähndrich, M., Leino, K. R. M., Müller, P., Schulte, W., and Venter, H. (2011). Specification and verification: the Spec# experience. *Communications of the ACM*, 54(6):81–91.

Barras, B., Boutin, S., Cornes, C., Courant, J., Filliatre, J.-C., Gimenez, E., Herbelin, H., Huet, G., Munoz, C., Murthy, C., et al. (1997). The coq proof assistant reference manual: Version 6.1.

Bjørner, D. and Jones, C. B., editors (1978). *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer.

Bjørner, D. and Jones, C. B. (1982). *Formal Specification and Software Development*. Prentice Hall International. ISBN 0-13-880733-7.

Bobot, F., Filliâtre, J.-C., Marché, C., and Paskevich, A. (2011). Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland.

De Moura, L. and Bjørner, N. (2008). Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer.

Europa Clipper Mission (2015). http://www.jpl.nasa.gov/missions/europa-mission.

Fitzgerald, J., Larsen, P. G., Mukherjee, P., Plat, N., and Verhoef, M. (2005). *Validated Designs For Object-oriented Systems*. Springer-Verlag TELOS, Santa Clara, CA, USA.

Fortress (2015). https://projectfortress.java.net/.

George, C., Haff, P., Havelund, K., Haxthausen, A., Milne, R., Nielsen, C. B., Prehn, S., and Wagner, K. R. (1992). *The RAISE Specification Language*. The BCS Practitioner Series, Prentice-Hall, Hemel Hampstead, England.

Gurevich, Y., Rossman, B., and Schulte, W. (2005). Semantic essence of AsmL. *Theoretical Computer Science*, 343(3):370–412.

Havelund, K. (2011). Closing the gap between specification and programming: VDM^{++} and Scala. In Korovina, M. and Voronkov, A., editors, *HOWARD-60: Higher-Order Workshop on Automated Runtime Verification and Debugging*, volume 1 of *EasyChair Proceedings*. Manchester, UK.

Jackson, D. (2012). *Software Abstractions: Logic, Language, and Analysis*. The MIT Press.

Jackson, E. K., Seifert, D., Dahlweid, M., Santen, T., Bjørner, N., and Schulte, W. (2009). Specifying and composing non-functional requirements in model-based development. In *Software Composition*, pages 72–89. Springer.

Jones, C. B. (1990). *Systematic Software Development using VDM*. Prentice Hall. ISBN 0-13-880733-7.

Jones, C. B. and Shaw, R. C., editors (1990). *Case Studies in Systematic Software Development*. Prentice Hall International. ISBN 0-13-880733-7.

Jones, S. L. P. (2003). *Haskell 98 language and libraries: the revised report*. Cambridge University Press.

K (2015). http://www.theklanguage.com.

Kim, S.-K., Burger, D., and Carrington, D. (2005). An mda approach towards integrating formal and informal modeling languages. In *FM 2005: Formal Methods*, pages 448–464. Springer.

Lausdahl, K., Lintrup, H. K. A., and Larsen, P. G. (2009). Connecting uml and vdm++ with open tool support. In *FM 2009: Formal Methods*, pages 563–578. Springer.

Leavens, G. T., Baker, A. L., and Ruby, C. (1998). Jml: a java modeling language. In *Formal Underpinnings of Java Workshop (at OOPSLA'98)*, pages 404–420. Citeseer.

Leino, R. (2010). Dafny: An automatic program verifier. In Clarke, E. M. and Voronkov, A., editors, *16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR 2010, Dakar, Senegal, April 25–May 1*, volume 6355 of *LNCS*. Springer.

MagicDraw (2015). https://www.magicdraw.com/.

Meyer, B. (1988). Eiffel: A language and environment for software engineering. *Journal of Systems and Software*, 8(3):199–246.

Milner, R., Tofte, M., and Harper, R., editors (1997). *The Definition of Standard ML*. MIT Press. ISBN 0-262-63181-4.

Nipkow, T., Paulson, L. C., and Wenzel, M. (2002). *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media.

OCaml (2015). http://caml.inria.fr/ocaml/index.en.html.

Odersky, M. (2010). Contracts for Scala. In *Runtime Verification - First Int. Conference, RV'10, St. Julians, Malta, November 1-4, 2010. Proceedings*, volume 6418 of *LNCS*, pages 51–57. Springer.

OMG (2012). Systems Modeling Language (SysML). http://www.omg.org/spec/SysML/1.3/.

OMG (2015). UML. http://www.omg.org/spec/UML/2.5/.

Owre, S., Rushby, J. M., , and Shankar, N. (1992). PVS: A prototype verification system. In Kapur, D., editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY. Springer-Verlag.

PVS (2015). http://pvs.csl.sri.com.

Python (2015). http://www.python.org.

Scala (2015). http://www.scala-lang.org.

SMT-LIB (2015). http://smtlib.cs.uiowa.edu.

Spivey, J. M. (1988). *Understanding Z: A Specification Language and Its Formal Semantics*. Cambridge University Press, New York, NY, USA.