Verified Change*

Klaus Havelund and Rahul Kumar

Jet Propulsion Laboratory
California Institute of Technology
California, USA

Abstract. We present the textual wide-spectrum modeling and programing language K, which has been designed for representing graphical SysML models, in order to provide semantics to SysML, and pave the way for analysis of SysML models. The current version is supported by the Z3 SMT theorem prover, which allows to prove consistency of constraints. The language is intended to be used by engineers for designing space missions, and in particular NASA's proposed mission to Jupiter's moon Europa. One of the challenges facing software development teams is the notion of change: the fact that code changes over time, and the subsequent problem of demonstrating that no harm has been done due to a change. K is in this paper being applied to demonstrate how change can be perceived as a software verification problem, and hence verified using more traditional software verification techniques.

Keywords: Modeling, programming, constraints, refinement, verification, change.

1 Introduction

The core topic of this paper is the concept of *change*, and how it relates to the way we *model* as well as *program* our systems, and how we can ensure correctness of change using modern verification technology. We shall specifically discuss this topic by introducing the wide-spectrum modeling and programming language K, under development at NASA's Jet Propulsion Laboratory (JPL), and demonstrate change scenarios and their verification in K.

The first call for opinion statements on the topic of change, with this journal in mind, was published in [48], in which it is characterized as "*a discipline for rigorously dealing with the nature of today's agile system development, which is characterized by unclear premises, unforeseen change, and the need for fast reaction, in a context of hard to control frame conditions, like third party components, network-problem, and attacks*". Our view is that change fundamentally can be considered as a software verification problem, where the question is the following: given a program P_1, and a new program P_2, does P_2 implement/refine P_1?

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

As such, in the extreme, the topic of correctness under change can potentially be considered as the well known topic of correctness.

As is well known, analysis of correctness can be performed dynamically by analyzing execution traces. Dynamic methods include such topics as testing, runtime verification, and specification mining, where learned models of previous behavior are compared to models of current behavior (after change). Dynamic methods are known to scale well, which is their attraction, but are also known to yield false negatives: not to report errors that exist, and in some cases (i.e. in dynamic analysis of data races and deadlocks) to yield false positives (to report errors that do not exist). Our focus in this paper is on static analysis of code: where proofs are carried out on the basis of the structure of the code.

The K language is being developed at NASA's Jet Propulsion Laboratory (JPL), as part of a larger effort at JPL to develop an in-house systems modeling language and associated tool set, referred to as EMS (Engineering Modeling System). EMS is based on the graphical SysML formalism [49], which is a variant of UML [50], both 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. EMS is being developed (and already in use) to support the design of NASA's proposed mission to Jupiter's moon Europa, referred to as the *Europa Clipper mission* [13], planned to launch around 2022 at the moment of writing.

The K language design was initially triggered by a desire from within the EMS project to create a textual version of SysML. Although the graphical nature of SysML can be attractive from a readability point of view, it suffers from a number of issues, including lack of clear semantics, lack of analysis capabilities, requiring heavy mouse-movement and clicking, and generally demanding much more visual space than a textual formalism. The K language is in addition inspired by the idea of combining modeling and programming into one language. This idea has been, and is being, pursued by others in various forms, including past works on wide-spectrum specification languages. See the related work section (Section 4) for a more detailed discussion. It is interesting to observe, that modern programming languages to an increasing degree look and feel like specification languages from the past.

A constantly returning discussion point in the design of K has been whether to actually design a new language or adopt an existing specification or programming language. The decision to design a new language was in part influenced by the felt need (perhaps unwarranted) to control the syntax and the tool development stack, including parsers, compilers, etc. Whether this decision was/is correct, an important factor that makes K interesting is that it is being designed to meet the needs of a real space mission. As such it can be considered as a confirmation of already existing work in the area of specification as well as programming languages.

The contents of the paper is as follows. Section 2 introduces the K language, focusing on a small analyzable subset, which is currently being tested during the

initial design phase of the Europa Clipper mission. Section 3 discusses various aspects of change by formalizing them in K. Section 4 outlines related work, and Section 5 concludes the paper. Appendix A contains the grammar for K in ANTLR [3] format.

2 The K Language

K is a textual language with various constructs for modeling and programming. The UML and SysML communities use the term *modeling*, whereas the formal methods community normally uses the term *specification*. We consider these terms for equivalent in this context, and shall use them interchangeably. A model (specification) is an abstract representation of a system (be it physical, conceptual, software, etc.), which has a concrete implementation, which in the software context is the program. It can be a physical object, however. The primary intended use for K is to easily create models and in the software context: implementations, and then be able to perform analysis on them. We primarily see K being used by system modelers who are used to expressing their models in SysML/UML. In this section, we briefly provide an overview of the K language. The presentation is centered around the example K model of geometric shapes, shown in Figure 1.

Classes : Similar to classes in other object-oriented languages, the class in K is the construct for performing abstraction. It is K's module concept. Classes can be arranged in packages, as in Java. A class can contain properties (corresponding to fields in Java), functions, and constraints, as discussed below. For example, class *Triangle* contains three properties, which are of type *TAngle*.

Inheritance : K provides the `extends` keyword for specifying an inheritance relation. In Figure 1, the *TAngle* class extends the *Angle* class. As a result, *TAngle*, not only inherits the properties and functions of *Angle*, but also the constraints. K also allows for multiple inheritance. Property and function names must be uniquely specified.

Primitive Types : K provides the following primitive types: `Int`, `Real`, `Bool`, `String`, `Char`, `Unit`.

Collections : K provides `Set`, `Seq`, and `Bag` as the three basic collections. K also provides support for tuples. The shapes model does not contain collections.

Properties : In K, properties can be present within classes or at the outermost level. Each property must have a name and a type. Our use of the term *property* is due to the use of this term in the model-based engineering (UML/SysML) community for name-type pairs, which in programming language and formal methods terminology normally are called constant/variable/field declarations. In the model shown in Figure 1, class *Shape* contains a single property named *sides* of type `Int`.

Modifiers : Each property can also have one or more *modifiers* specified for it, for example **val**/**var** to make the property read only or writable (the default being read only). The shapes model does not contain modifiers.

```
class Shape { sides : Int }

class Angle {
  value : Int

  fun eq(other: Angle) : Bool { value = other.value }

  req value >= 0 && value <= 360
}

class TAngle extends Angle {
  req value < 180
}

class Triangle extends Shape {
  a : TAngle
  b : TAngle
  c : TAngle

  req sides = 3
  req Angles: a.value + b.value + c.value = 180
}

class  Equilateral extends Triangle {
  req a.eq(b) && b.eq(c)
}

class Obtuse extends Triangle {
  req a.value > 90 || b.value > 90 || c.value > 90
}
```

Fig. 1: A simple model of geometrical shapes in K

Constraints : K provides syntax for specifying constraints in a class. This is done using the `req` keyword (we use the term *requirements* for constraints) followed by a name (optional), and an expression that specifies the constraint on the class. Constraints are class invariants. However, since side effects are not supported with analysis in the current version of K, constraints simply become axioms on names defined in the signature of a class. For example, in class *Angle*, the *value* of any angle should always be between 0 and 360 degrees. Multiple constraints can also be specified. Their effect is the same as if all expressions were conjoined into a single constraint. For example, each instance of the *Triangle* class should have exactly three sides and the sum of the angles should be exactly 180 degrees. Constraints are expressed in predicate logic, including universal and existential quantification.

Functions : A function provides the ability to perform computation. In K, functions can take arguments and return the result of the computation of the function. The *eq* function in class *Angle* compares a given argument angle's value to the class local angle value, returning a `Bool`.

Function Specifications : Each function in K can also have a *specification* associated with it. The specification can be a `pre`-condition and/or a `post`-condition. The shapes example does not contain function specifications. They are discussed in more detail along with an example in Section 3.

Expression Language : Similar to other high-level specification languages, K provides a rich expression language for specifying functions and requirements. K generally provides predicate logic with multiple operators, such as arithmetic operators, Boolean operators, if-then-else, blocks (a form of let-expressions inspired by Scala), set and sequence constructors and operators, and universal and existential quantification.

Annotations K provides the ability to create new annotations by specifying a name and a type for the annotation. The annotations can then be applied by writing an @ sign followed by the annotation name, and possible parameters, immediately before the element that is desired to be annotated. There is no limit on how many annotations can be applied to any entity.

Comments : Single line comments can be specified with the prefix −− and multi-line comments are specified with === as both the start and end of the multi-line comment.

In addition to the language constructs described here, K also has syntactic support for programming with *side effects* (assignment, sequential composition, and looping constructs, although these concepts are not yet supported by theorem proving or execution), *type abbreviations*, as well as SysML/UML specific concepts such as *associations* and *multiplicities* (which can be used for specifying the allowed size of a property). To conserve space, these constructs are not discussed in further detail. Appendix A shows part of the K ANTLR grammar, omitting grammar rules dealing with parsing of values of primitive types, such as digits, strings, etc.

Currently, the K infrastructure comes with a parser, that has been generated using ANTLR version 4. Using the parser, an abstract syntax tree is created that is used for performing type checking. In addition, we have also developed a translator from K to SMT2, which currently is processed by the Z3 theorem prover [11]. This is used as a means to perform various checks such as function specification satisfiability, class consistency checking, and model generation. More generally, K works (similarly to Z3) by finding an assignment to all properties that satisfies the requirements (constraints). In case there are more than one assignment satisfying the requirements, an arbitrary is returned (although the result is deterministic). In case no assignment is possible, an identification of which requirements are inconsistent is returned. In the worst case scenario, the solver cannot determine the result in reasonable time, and it times out after a user-definable time period. The entire K infrastructure is implemented using Scala. It is planned to make a subset of the language executable. The source

code along with binary releases along with a fully functional online solver and K editor can be found at [33]. It is used by engineers and developers at JPL for expressing requirements and verifying their consistency. The use is, however still in evaluation mode.

3 Change

3.1 Behavior

K at JPL so far, has mostly been used for specifying static structure, similar to what can be represented by UML/SysML class diagrams, with requirements typically constraining integer and real variables that represent properties of physical nature, such as for example weights and distances. The shapes model in Figure 1 is an example of a model of structure, namely the shape of triangles. K can, however, also be used for specifying behavior, using the same concepts used for specifying structure, namely classes, properties, functions and requirements on these. The idea is in other words to use mathematical logic to represent behaviors. This is an illustration of the pursued objective during the design of K to keep the language as small as possible, relying as much as possible on the language of mathematics for expressing problems and solutions. This approach of course has algorithmic consequences when it comes to analyzing models. Our intention is to stay in mathematics as far as the tooling (existing theorem provers) allows us. In the following subsections we illustrate how one can encode two different behavior concepts in K, namely state machines and event scheduling.

State Machines State machines are commonly used to specify the behavior of software as well as hardware systems. They are frequently used at JPL for specifying the behavior of embedded flight software modules controlling, for example, planetary rovers. A state machine is defined by a collection of states, a collection of events, and a labeled transition relation (labels are events) between states. This can of course be modeled in numerous ways. Here we shall assume deterministic state machines, and model the transition relation as a function. The K model in Figure 2 represents an encoding of a state machine modeling a rocket engine, which can be in one of the states: *off*, *ready* or *firing*. Events include *turn_on*, *fire*, and *turn_off*. The types of states and events (*State* and *Event*) are modeled as body-less classes. The class *RocketEngine* models our state machine. It defines the three states as well as the three events as properties of the appropriate types. A requirement expresses that the states are all different (a similar requirement should in principle also be provided for events).

The function *move* represents the transition relation, and is declared to take two arguments: a state and event, and to return a state. It has no body, meaning that it is yet to be defined. The subsequent four requirements define the *move* function. For example the first requirement states that in the *off* state, on encountering a *turn_on* event, the engine moves to the *ready* state.

```
class State
class Event

class RocketEngine {

  off : State
  ready : State
  firing : State

  turn_on : Event
  fire : Event
  turn_off : Event

  req off != ready && off != firing && firing != ready

  fun move(s: State, e: Event) : State

  req move(off, turn_on) = ready
  req move(ready, fire) = firing
  req move(firing, turn_off) = ready

  req move(ready, fire) = off -- added Monday morning
}
```

Fig. 2: State machine

The last requirement demanding the engine to move from the *ready* state to the *off* state on a *fire* event was added on a Monday morning by a tired modeler, and in our context represents a change to the model. This requirement, however, is inconsistent with a previous requirement that demands the resulting state to be *firing*. Since *move* is a function (the transition relation is deterministic), and cannot return two different values for the same argument, this is detected by the solver. Without this last requirement, the solver will declare the model satisfiable, and will synthesize the state machine function based on the provided requirements. Note, however, that not all transitions are modeled, hence the synthesized state machine may not be the desired one.

Event Scheduling Event scheduling is a very common problem faced in a plethora of fields and domains. The typical scenario usually involves specifying multiple different allowed orderings of events and determining whether the specified ordering is satisfiable or not, and if satisfiable, generating a timeline for the specified events. Figure 3 shows one such example encoded using K.

The foundation of the scheduling problem is specified by the *Event* class, which represents a single event that has a start and end time (both of these

specified using the `Int` type). The requirements on this class specify that the duration of any event has to be non-zero (`req` *nonZeroDuration*) and the start time of all events is greater than or equal to zero (`req` *afterBigBang*). Further, the three functions in the *Event* class encode Allen logic operators [2] using simple mathematical expressions on the start and end times of two events in question. The functions specify whether an event occurs *before* another event e, *meets* another event e, or *contains* another event e. We do not include the full set of Allen logic operators for sake of brevity.

```
class Event {
  startTime : Int
  endTime : Int

  req afterBigBang : startTime >= 0
  req nonZeroDuration : endTime > startTime

  fun before (e : Event) : Bool { endTime < e.startTime }

  fun meets (e : Event) : Bool { endTime = e.startTime }

  fun contains (e : Event) : Bool {
    (startTime <= e.startTime && e.endTime < endTime) ||
    (startTime < e.startTime && e.endTime <= endTime)
  }
}

class Schedule {
  sciWin  : Event
  digHole : Event
  takePic : Event
  commWin : Event
  config  : Event
  comm    : Event

  oneScienceActivity : Bool =
    (sciWin.contains(digHole) || sciWin.contains(takePic)) &&
   !(sciWin.contains(digHole) && sciWin.contains(takePic))

  req  sciWin.before(commWin) && oneScienceActivity &&
       commWin.contains(config) && commWin.contains(comm) &&
       config.meets(comm)
}
```

Fig. 3: Scheduling

Using this foundation, it is now easy to specify schedules and events. This is exactly what the *Schedule* class does. Five events are created as instances of the *Event* class. The actual schedule is specified as a requirement, which expresses the ordering of the events. In this particular example, the schedule specifies that the *sciWindow* (science window) must occur before the *commWindow* (communication window) and at least one science activity must have taken place. By translating the scheduling specification to SMT2 and applying the Z3 theorem prover, it can now be checked if the provided ordering on the events is satisfiable or not by checking if the *Schedule* class is satisfiable. A satisfying assignment to the class also provides us with a concrete time line of the events. It can be common for schedules and events to change as the project evolves and reaches maturity. Using such a mechanism for encoding the schedule of events provides great power and flexibility. Dealing with the changing schedule is easily done by either modifying existing requirements in the *Schedule* class, or by adding new events and requirements. Each change in the schedule is also verified using a theorem prover, which adds greater confidence in the change.

3.2 Refinement

Change can be considered as refinement. In the formal methods literature refinement usually refers to the situation where one model/program, the specification, and typically abstract of nature, is replaced by a lower level model/program, the implementation. Along with the refinement normally goes a proof, that the implementation refines the specification. The literature offers many solutions to how specifications, implementations and refinements are expressed as well as proved correct, see for example [6, 7, 31, 16, 17]. We shall not here enumerate all of these, but bring forward two examples, one illustrating function refinement, and one illustrating data refinement.

Function Refinement Function refinement consists of making the body of a function more concrete, while the signature (name as well as argument and result types) of the function stays unchanged. More generally, data structures accessed by the function stay unchanged. One popular approach to this is design-by-contract, where a function is first specified using pre/post conditions, and then later implemented with a function body. This form of refinement is advocated for example in specification languages such as VDM [6, 7, 31] and RAISE [16, 17], as well as in programming languages such as Eiffel [12] and Java in the form of the JML comment language [30].

K supports design-by-contract using pre/post conditions. The example in Figure 4 illustrates this with two class definitions. The left-most class *Util_Spec* represents the specification of a mathematics utility module containing two functions, *min* for computing the minimum of two values, and *abs* for returning the absolute value of an integer. Both functions are specified with a post condition stating what is expected to be true about the resulting value, denoted by *$result*. As an example, the post condition for the *min* function states that the result is equal to one of the arguments, and it is smallest such.

Specification

```
class Util_Spec {
  fun min(x:Int,y:Int): Int
    post
      ($result = x ||
       $result = y) &&
       $result <= x &&
       $result <= y

  fun abs(x:Int):Int
    post $result >= 0 &&
         ($result = x ||
          $result = -x)
}
```

Implementation

```
class Util extends Util_Spec {
  fun min(x:Int,y:Int): Int {
    if x <= y then x else y
  }

  fun abs(x:Int):Int {
    if x < 0 then -x else x
  }
}
```

Fig. 4: Mathematical function refinement

The class *Util* to the right extends class *Util_Spec* and refines the functions with proper function bodies. The semantics of K is such that the refined function bodies will have to satisfy the post conditions. The K solver proves this automatically in this case. The fact that the implementation class extends (inherits from) the specification class reflects that this form of refinement is a form of theory refinement, where the theory denoted by the implementation must imply that of the specification: the implementation signature contains that of the specification, and the requirements logically imply those of the specification:

$$\mathit{Implementation} \Rightarrow \mathit{Specification}$$

Data Refinement Data refinement consists of changing the data structures used, which will cause functions to change as well. Data refinement has for example been advocated in the VDM method [6, 7, 31], which is the approach we shall illustrate here using K. The approach consists of defining a specification and an implementation as follows. The specification consists of a type Σ_a of abstract states, as well as abstract operations $opn_a : \Sigma_a \to \Sigma_a$ on this state. The implementation consists of a type Σ_c of concrete states, as well as concrete operations $opn_c : \Sigma_c \to \Sigma_c$ on this state. To perform a proof of correctness of the refinement, an abstraction function $abs : \Sigma_c \to \Sigma_a$ from the type of concrete implementation states to the type of abstract specification states must be provided, and the following property must (amongst others) be proved for each operation opn, where opn_a is the abstract version and opn_c is the concrete

version, and $pre_a : \Sigma_a \to \mathbb{B}$ is the pre-condition of the abstraction operation opn_a:

$$\forall \sigma : \Sigma_c \cdot pre_a(abs(\sigma)) \Rightarrow opn_a(abs(\sigma)) = abs(opn_c(\sigma)) \quad (1)$$

Each concrete operation must in other words be proved to update the concrete state in a manner corresponding to the desired operation on the abstract state. We illustrate this approach with a rather simple K model of a light switch, which can be turned on and off. The specification is shown on the left of Figure 5. A state is defined abstractly as an object of a class *State*. The two states *off* and *on* are defined as distinct states of that type. Two functions are defined, one for toggling the state (*toggle*), and one for testing whether the light switch is on (*isOn*). The *toggle* function is only declared by its signature, no function body is provided. The behavior is instead provided as a couple of requirements.

Specification

```
class State

class LightSwitch_Spec {
  off : State
  on  : State

  req off != on

  fun toggle(s:State):State

  fun isOn(s:State):Bool {
    s = on
  }

  req toggle(off) = on
  req toggle(on) = off
}
```

Implementation

```
class LightSwitch {
  fun toggle(cs: Int): Int {
    if cs = 1 then 0 else 1
  }

  fun isOn(cs: Int): Bool {
    cs = 1
  }
}
```

Fig. 5: Lightswitch refinement

The implementation is shown on the right of Figure 5. Here we have decided to model the state as an integer, being 1 when the light switch is on and 0 when it is off. Note that in this case the implementation does not extend (inherit from) the specification as was the case in the mathematical function refinement in Figure 4. Instead, the proof corresponding to equation (1) above is provided

```
class RefinementProof {
  spec : LightSwitch_Spec
  impl : LightSwitch

  fun abs(cs:Int):State {
    if cs = 1 then spec.on else spec.off
  }

  req forall cs: Int :-
    spec.toggle(abs(cs)) = abs(impl.toggle(cs))

  req forall cs: Int :-
    spec.isOn(abs(cs)) = impl.isOn(cs)

}
```

Fig. 6: Lightswitch refinement proof

in the separate class *RefinementProof* in Figure 6. To express the refinement property to be proved, an instance *spec* of the specification and an instance *impl* of the implementation are created such that we can refer to their respective operations (functions). Then the abstraction function *abs* is defined from the concrete state of integers to the abstract state *State* of the specification. Finally, the requirement is the K formulation of equation (1) above, ignoring the pre-condition part since all pre-conditions in this example are true. The K solver proves the implementation correct automatically. An incorrect modification of the implementation, such as for example to change the body of *isOn* in the implementation to $cs = 0$ will dually be caught by the solver.

4 Related Work

K is intended to represent a textual modeling language capable of representing SysML concepts, specifically class diagrams with constraints. However, as mentioned in the introduction, it also contains programming constructs, although these are not yet supported by theorem proving or execution. As such it can be perceived as a wide-spectrum modeling/programming language.

Wide-spectrum specification languages have been investigated to length in the formal methods community. One of the well-known examples is VDM [6, 7, 31, 32]. VDM in its original form [6] provided a combination of procedural programming and functional programming, as well as specification using sets, lists and maps (with proper mathematical notation), and higher-order predicate logic. VDM^{++} [14] added object-orientation to VDM, which is now part of the VDM standard. The RAISE specification language (RSL) [16] is a wide-spectrum language taking inspiration from VDM as well as from other modeling languages

such as Z [47], and algebraic equational specification languages. Here refinement is the simpler theory implication: the implementation shall imply the specification in a logic sense. AsmL [19] is a more recent wide-spectrum specification language, in many ways similar to VDM, but based on the idea that operations with side effects operate on algebras. Other fundamental works on refinement include (not a comprehensive list): [51, 25, 38, 52, 4, 1].

Alloy [29] added new life to this community by being supported by an automated SAT solver. In many respects, K is close in spirit to Alloy, but differs by being supported by an automated SMT solver (in contrast to a SAT solver), resulting in a richer set of constructs, including arithmetic, being exposed to analysis. K also combines a type view as found in traditional specification and programming languages, as well as a relational view, whereas Alloy is purely relational. We are of the belief that the notion of a type is fundamental to programming as well as to modeling. In contrast to automated provers, interactive theorem provers such as PVS [41, 43], Coq [10], and Isabelle [28], allow the user to steer the proofs. Although this allows to perform more complex proofs, it also requires more skills of the user, and time, which is often a limited resource in software development projects.

Several high-level programming languages have been developed over time, including the early SML (Standard ML) [37], its derivative Ocaml [39], and Haskell [21]. However, also Java can be considered high-level due to its libraries of collections (sets, lists, and maps), as well as the iterator concept. Python [44] is close to combining object-oriented and functional programming. Scala [45] does this to the full extent. The close relationship between Scala and VDM is discussed in [22]. Fortress [15] introduced built-in notation for sets, lists, and maps, very much resembling the notation in VDM.

Specification constructs have been introduced in programming languages, in the form of design-by-contract (pre/post conditions + class invariants). Examples are Eiffel [12] and Spec$^{\#}$ [46], where contracts are part of the language. Scala has library functions for writing pre/post conditions on functional programs [40]. Finally, The JML language [30] allows to write design-by-contract specifications for Java as comments. These are ignored by the standard Java compiler, and therefore must be processed with special tools. EML (Extended ML) [34] takes a slightly different approach to specification and formal development of SML programs. EML specifications look just like SML programs except that axioms are allowed in signatures and in place of code in structures and functors. Some EML specifications are executable, since SML function definitions are just axioms of a certain special form. This makes EML a wide-spectrum language.

Programming languages are now also being designed with verification in mind. Dafny [36] supports specifications that can be used to write correctness conditions for programs. It is supported by a verifier, which is implemented on top of the Boogie verification engine, which itself is built on top of Z3. Why3 [8] provides a rich language for specification and programming, called WhyML, and relies on external theorem provers, both automated and interactive, to discharge verification conditions. A user can write WhyML programs directly and

get correct-by-construction Ocaml programs through an automated extraction mechanism. Model checking is another form of analysis that has been applied to programming languages. Java PathFinder [23, 24] performs model checking of Java programs. SLAM [5] performs static analysis and counter-example guided abstraction refinement of device drivers, and has been applied in a large scale industry setting. Spin [27] performs model checking of models expressed in the Promela language, but can also model check C code directly. The ABS [20] language and system provide various types of analysis such as resource analysis, deadlock analysis, as well as tools to perform test generation and formal verification. The notions of abstract contracts and abstract class invariants are introduced in [9], in order to reduce proof efforts when contracts change. In [26] is presented an approach to integrate a semiautomatic verification tool into a state-of-the-art integrated development environment (IDE), with the specific objective to keep implementation, specification and proofs in sync.

The great improvements in model checking, static analysis, theorem proving, and SMT solvers such as Z3 have all contributed to investigating and dealing with software change. To this effect, differential symbolic execution [42] has been investigated for establishing equivalence between two versions of a program. The work described in [35] uses verification conditions and SMT solvers for detecting semantic change between two closely related versions of a function (program), by discovering inputs to the function that cause the outputs to differ. The work described in [18] deals with regression verification and provides a technique for performing equivalence checking of C programs, by using the older version of the program as a specification for the new version of the program. A large part of the inspiration for such work comes from the theorem proving community.

An important use of K that we have observed so far, which differs in the way traditional verification tools are used, is that modelers tend to use K along with it's solving ability as a tool for *discovering* the right set of requirements for their class before introducing a change. For example, uncertainty about a particular variable and it's potential range of valid values can be quite common in modeling environments. Since K helps discover unsatisfiability, modelers use an iterative refinement technique to discover the appropriate range of a variable for their needs. K in this case is providing validation before a change is completely committed.

5 Conclusion

In this paper, we have addressed the topic of *change* in a software/modeling development environment. More generally, we have developed what we refer to as a *development language* for modeling as well as programming, also referred to as a wide-spectrum programming language, with verification support. This enables developers to easily study properties of their models and programs, and in particular, in this case, the effect of their change, thus helping to avoid making changes that could potentially lead to unsatisfiability and inconsistencies. We have studied various scenarios, and how *consistency* checking and *change* viewed

as *refinement* can be applied to each of those. While the topic of change itself is extremely broad, we believe that a language oriented approach as presented in this paper provides concrete value and provides a good foundation for developing stronger techniques.

Acknowledgements. We would like to thank Chris Delp and Bradley Clement for the opportunities and insights they provided during the development of the K language.

References

1. J.-R. Abrial. *Modeling in Event-B*. Cambridge University Press, 2010.
2. J. F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23:123–154, 1984.
3. ANTLR. http://www.antlr.org.
4. R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer Verlag, 1998.
5. T. Ball, E. Bounimova, R. Kumar, and V. Levin. Slam2: Static driver verification with under 4% false alarms. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, pages 35–42. FMCAD Inc, 2010.
6. D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer, 1978.
7. D. Bjørner and C. B. Jones. *Formal Specification and Software Development*. Prentice Hall International, 1982. ISBN 0-13-880733-7.
8. F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011.
9. R. Bubel, R. Hähnle, and M. Pelevina. Fully abstract operation contracts. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part II*, volume 8803 of *LNCS*, pages 120–134. Springer, 2014.
10. Coq. https://coq.inria.fr.
11. L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
12. Eiffel. http://www.eiffel.com.
13. Europa Clipper Mission. http://www.jpl.nasa.gov/missions/europa-mission.
14. J. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef. *Validated Designs For Object-oriented Systems*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2005.
15. Fortress. http://java.net/projects/projectfortress.
16. C. George, P. Haff, K. Havelund, A. Haxthausen, R. Milne, C. B. Nielsen, S. Prehn, and K. R. Wagner. *The RAISE Specification Language*. The BCS Practitioner Series, Prentice-Hall, Hemel Hampstead, England, 1992.
17. C. George and A. Haxthausen. The logic of the RAISE specification language. In D. Bjørner and M. Henson, editors, *Logics of Specification Languages*, Monographs in Theoretical Computer Science, pages 349–399. Springer Berlin Heidelberg, 2008.
18. B. Godlin and O. Strichman. Regression verification. In *Proceedings of the 46th Annual Design Automation Conference*, pages 466–471. ACM, 2009.

19. Y. Gurevich, B. Rossman, and W. Schulte. Semantic essence of AsmL. *Theoretical Computer Science*, 343(3):370–412, 2005.
20. R. Hähnle. The abstract behavioral specification language: a tutorial introduction. In *International Symposium on Formal Methods for Components and Objects*, pages 1–37. Springer, 2012.
21. Haskell. http://www.haskell.org/haskellwiki/Haskell.
22. K. Havelund. Closing the gap between specification and programming: VDM^{++} and Scala. In M. Korovina and A. Voronkov, editors, *HOWARD-60: Higher-Order Workshop on Automated Runtime Verification and Debugging*, volume 1 of *EasyChair Proceedings*, December 2011. Manchester, UK.
23. K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer, STTT*, 2(4), April 2000.
24. K. Havelund and W. Visser. Program model checking as a new trend. *STTT*, 4(1):8–20, 2002.
25. J. He, C. Hoare, and J. Sanders. Data refinement refined. In *European Symposium on Programming*, volume 213 of *LNCS*. Springer, 1986.
26. M. Hentschel, S. Käsdorf, R. Hähnle, and R. Bubel. An interactive verification tool meets an IDE. In E. Albert and E. Sekerinski, editors, *Integrated Formal Methods - 11th International Conference, IFM 2014, Bertinoro, Italy, September 9-11, 2014, Proceedings*, volume 8739 of *LNCS*, pages 55–70. Springer, 2014.
27. G. J. Holzmann. *The Spin Model Checker – Primer and Reference Manual*. Addison-Wesley, 2004.
28. Isabelle. https://isabelle.in.tum.de.
29. D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2012.
30. JML. http://www.eecs.ucf.edu/ leavens/JML.
31. C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall, 1990. ISBN 0-13-880733-7.
32. C. B. Jones and R. C. Shaw, editors. *Case Studies in Systematic Software Development*. Prentice Hall International, 1990. ISBN 0-13-880733-7.
33. K. http://www.theklanguage.com.
34. S. Kahrs, D. Sannella, and A. Tarlecki. The definition of Extended ML: A gentle introduction. *Theoretical Computer Science*, 173:445–484, 1997.
35. S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *Computer Aided Verification*, pages 712–717. Springer, 2012.
36. R. Leino. Dafny: An automatic program verifier. In E. M. Clarke and A. Voronkov, editors, *16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR 2010, Dakar, Senegal, April 25May 1*, volume 6355 of *LNCS*. Springer, 2010.
37. R. Milner, M. Tofte, and R. Harper, editors. *The Definition of Standard ML*. MIT Press, 1997. ISBN 0-262-63181-4.
38. C. Morgan. *Programming from Specifications*. Prentice Hall, 1994. 2nd ed.
39. OCaml. http://caml.inria.fr/ocaml/index.en.html.
40. M. Odersky. 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, 2010.
41. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *LNAI*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

42. S. Person, M. B. Dwyer, S. Elbaum, and C. S. Psreanu. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 226–237. ACM, 2008.
43. PVS. http://pvs.csl.sri.com.
44. Python. http://www.python.org.
45. Scala. http://www.scala-lang.org.
46. Spec#. http://research.microsoft.com/en-us/projects/specsharp.
47. J. M. Spivey. *Understanding Z: A Specification Language and Its Formal Semantics*. Cambridge University Press, New York, NY, USA, 1988.
48. B. Steffen. LNCS transactions on foundations for mastering change: Preliminary manifesto. In T. Margaria and B. Steffen, editors, *6th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation, ISoLA 2014, Corfu, Greece, October 8-11*, volume 8803 of *LNCS*. Springer, 2014.
49. SysML. http://www.omgsysml.org.
50. UML. http://www.uml.org.
51. N. Wirth. Program development by stepwise refinement. *CACM: Communications of the ACM*, 14, 1971.
52. J. Woodcock and J. Davies. *Using Z. Specification, Refinement, and Proof*. Prentice-Hall, 1996.

A K Grammar

```
model:
  packageDeclaration? importDeclaration*
  annotationDeclaration* topDeclaration* EOF;

packageDeclaration: 'package' qualifiedName ;

importDeclaration: 'import' qualifiedName ('.' '*')? ;

annotationDeclaration: 'annotation' Identifier ':' type ;

annotation: '@' Identifier '(' expression ')' ;

topDeclaration: annotation* entityDeclaration | annotation* memberDeclaration ;

entityDeclaration:
  ('class'|'assoc'|Identifier) Keyword? Identifier typeParameters? extending?
  ('{' block '}')? ;

Keyword: '<' Identifier '>' ;

typeParameters: '[' typeParameter (',' typeParameter)* ']' ;

typeParameter: Identifier (':' typeBound)? ;

typeBound: type ('+' type)* ;

extending: 'extends' type (',' type)* ;

block: blockDeclaration* ;

blockDeclaration: annotation* memberDeclaration  ;

memberDeclaration:
 typeDeclaration | propertyDeclaration | functionDeclaration |
 constraint | expression  ;

typeDeclaration: 'type' Identifier (typeParameters? '=' type )? ;

propertyDeclaration:
  propertyModifier* Identifier ':' type multiplicity? (('='|':=') expression)? ;

propertyModifier:
  'part' | 'var' | 'val' | 'ordered' | 'unique' | 'source' | 'target';

functionDeclaration:
  'fun' Identifier typeParameters? ('(' paramList ')')? (':' type)?
  functionSpecification* ('{' block '}')? ;

paramList: param (',' param)* ;

param: Identifier ':' type ;

functionSpecification: 'pre' expression | 'post' expression ;

constraint: 'req' (Identifier ':')? expression ;

multiplicity: '[' expressionOrStar (',' expressionOrStar)? ']' ;

expressionOrStar: expression | '*' ;

type:
  primitiveType | classIdentifier typeArguments? | type ('*' type)+ |
  type '->' type | '(' type ')' | '{|' Identifier ':' type ':-' expression '|}' ;

primitiveType: 'Bool' | 'Char' | 'Int' | 'Real' | 'String' | 'Unit' ;
```

```
classIdentifier: qualifiedName | 'Class' | collectionKind ;

collectionKind: 'Set' | 'Bag' | 'Seq' ;

typeArguments: '[' type (',' type)* ']' ;

expression:
    '(' expression ')' | 'Tuple' '(' expression (',' expression)+ ')'
  | literal | Identifier | expression '.' Identifier
  | expression '(' argumentList? ')'
  | '!' expression | '{' block '}'
  | 'if' expression 'then' expression ('else' expression)?
  | 'match' expression 'with' match+
  | 'while' expression 'do' expression
  | 'for' pattern 'in' expression 'do' expression
  | collectionKind '{' expressionList? '}'
  | collectionKind '{' expression '..' expression '}'
  | collectionKind '{' expression '|' rngBindingList ':-' expression '}'
  | expression ('*'|'/'|'%'|'inter'|'\\'|'++'|'#'|'^') expression
  | expression ('+'|'-'|'union') expression
  | expression ('<='|'>='|'<'|'>'|'=') expression
  | expression ('!='|'isin'|'!isin'|'subset'|'psubset') expression
  | expression ('&&'|'||') expression
  | expression ('=>'|'<=>') expression
  | expression (':='|'is'|'as') expression
  | 'assert' '(' expression ')'
  | '-' expression
  | qualifiedName '~'
  | 'forall' rngBindingList ':-' expression
  | 'exists' rngBindingList ':-' expression
  | pattern '->' expression
  | 'continue' | 'break'  | 'return' expression? | '$result' ;

match: 'case' pattern ('|' pattern)* '=>' expression ;

argumentList: positionalArgumentList | namedArgumentList ;

positionalArgumentList: expression (',' expression)* ;

namedArgumentList: namedArgument (',' namedArgument)* ;

namedArgument : Identifier '::' expression ;

collectionOrType: expression | type ;

rngBindingList: rngBinding (',' rngBinding)* ;

rngBinding: patternList ':' collectionOrType ;

patternList: pattern (',' pattern)* ;

pattern:
  literal | '_' | Identifier | '(' pattern (',' pattern)+ ')' | pattern ':' type ;

identifierList: Identifier (',' Identifier)* ;

expressionList: expression (',' expression)* ;

qualifiedName: Identifier ('.' Identifier)* ;

literal:
  IntegerLiteral | RealLiteral | CharacterLiteral | StringLiteral |
  BooleanLiteral | NullLiteral | ThisLiteral ;
```