Generated Scala Code for Attack Trees

Florian Kammüller

Middlesex University London and Technische Universität Berlin

Abstract

This paper contains the Scala code that has been automatically generated from the Attack Tree formalisation of the Isabelle Infrastructure [1].

Keywords: Attack trees, Formal methods, Verification, Probability, Quantum Cryptography

1. Scala Code for Validity Predicate

This appendix describes the Scala code generated from the Isabelle theories for the automatic computation of the attack tree predicate `is_attacktree`.

We generate executable code for this test predicate in the programming language Scala using Isabelle's code generation mechanism [2]. Note that Scala is a functional programming language which runs on the Java virtual machine. As such Scala is a language that is particularly interesting for industrial applications, since it shares the property of Java that it is platform independent; furthermore it is compatible with Java in that it is possible to integrate Scala programs with Java programs. The extraction mechanism in Isabelle allows to transform computational definitions in Isabelle to functions in Scala. As a consequence, the Scala code shares its properties with the corresponding definitions in Isabelle and these properties are formally guaranteed by the proofs in the Isabelle system.

The code generation mechanism is very easy to evoke for executable parts of an Isabelle specification. We simply added the following line after the predicate's definition.

```
export_code is_attack_tree in Scala
```

The code is instantaneously generated and output within the Isabelle interface.

More than half of the code has been generated to implement the datatypes used in the predicate's definition: the objects `HOL`, `ProductType`, `Lista`, `Set` contain implementations of equality, pairs, lists, and sets generated from the corresponding formalizations in the Isabelle theory library. The object `MC` contains the code generated from the Isabelle theory `MC.thy` for the abstract state

Email address: `f.kammueller@mdx.ac.uk` (Florian Kammüller)

transition. The concept of a type class in Isabelle is mapped by the code generation to the concept of a trait in Scala. Similarly, the other concepts of Isabelle's specification language, like datatypes, recursive function definitions, polymorphic types or pattern matching are mapped quite simply to equivalent concepts in Scala. On inspecting the code of the object `AT` after getting used to different syntax it is quite apparent that this function `is_attack_tree` is equal to the one in Section **??**.

1.1. HOL and Product Type

```
object HOL {
trait equal[A] {
  val ‘HOL.equal‘: (A, A) => Boolean
}
def equal[A](a: A, b: A)(implicit A: equal[A]): Boolean = A.‘HOL.equal‘(a, b)
object equal {
  implicit def ‘Set.equal_set‘[A : equal]: equal[Set.set[A]] = new
    equal[Set.set[A]] {
    val ‘HOL.equal‘ = (a: Set.set[A], b: Set.set[A]) => Set.equal_seta[A](a, b)
  }
}
def eq[A : equal](a: A, b: A): Boolean = equal[A](a, b)
} /* object HOL */

object Product_Type {
def fst[A, B](x0: (A, B)): A = x0 match {
  case (x1, x2) => x1
}
def snd[A, B](x0: (A, B)): B = x0 match {
  case (x1, x2) => x2
}
def equal_prod[A : HOL.equal, B : HOL.equal](x0: (A, B), x1: (A, B)): Boolean =
  (x0, x1) match {
  case ((x1, x2), (y1, y2)) => HOL.eq[A](x1, y1) && HOL.eq[B](x2, y2)
}
} /* object Product_Type */
```

1.2. Lists

```
object Lista {
def fold[A, B](f: A => B => B, x1: List[A], s: B): B = (f, x1, s) match {
  case (f, x :: xs, s) => fold[A, B](f, xs, (f(x))(s))
  case (f, Nil, s) => s
}
def nulla[A](x0: List[A]): Boolean = x0 match {
  case Nil => true
  case x :: xs => false
}
def filter[A](p: A => Boolean, x1: List[A]): List[A] = (p, x1) match {
  case (p, Nil) => Nil
```

```
  case (p, x :: xs) => (if (p(x)) x :: filter[A](p, xs) else filter[A](p, xs))
}
def member[A : HOL.equal](x0: List[A], y: A): Boolean = (x0, y) match {
  case (Nil, y) => false
  case (x :: xs, y) => HOL.eq[A](x, y) || member[A](xs, y)
}
def insert[A : HOL.equal](x: A, xs: List[A]): List[A] =
  (if (member[A](xs, x)) xs else x :: xs)
def hd[A](x0: List[A]): A = x0 match {
  case x21 :: x22 => x21
}
def tl[A](x0: List[A]): List[A] = x0 match {
  case Nil => Nil
  case x21 :: x22 => x22
}
def list_ex[A](p: A => Boolean, x1: List[A]): Boolean = (p, x1) match {
  case (p, Nil) => false
  case (p, x :: xs) => p(x) || list_ex[A](p, xs)
}
def removeAll[A : HOL.equal](x: A, xa1: List[A]): List[A] = (x, xa1) match {
  case (x, Nil) => Nil
  case (x, y :: xs) =>
    (if (HOL.eq[A](x, y)) removeAll[A](x, xs) else y :: removeAll[A](x, xs))
}
def list_all[A](p: A => Boolean, x1: List[A]): Boolean = (p, x1) match {
  case (p, Nil) => true
  case (p, x :: xs) => p(x) && list_all[A](p, xs)
}
} /* object Lista */
```

1.3. Sets

```
object Set {
abstract sealed class set[A]
final case class seta[A](a: List[A]) extends set[A]
final case class coset[A](a: List[A]) extends set[A]
def member[A : HOL.equal](x: A, xa1: set[A]): Boolean = (x, xa1) match {
  case (x, coset(xs)) => ! (Lista.member[A](xs, x))
  case (x, seta(xs)) => Lista.member[A](xs, x)
}
def less_eq_set[A : HOL.equal](a: set[A], b: set[A]): Boolean = (a, b) match {
  case (coset(Nil), seta(Nil)) => false
  case (a, coset(ys)) => Lista.list_all[A](((y: A) => ! (member[A](y, a))), ys)
  case (seta(xs), b) => Lista.list_all[A](((x: A) => member[A](x, b)), xs)
}
def equal_seta[A : HOL.equal](a: set[A], b: set[A]): Boolean =
  less_eq_set[A](a, b) && less_eq_set[A](b, a)
def Bex[A](x0: set[A], p: A => Boolean): Boolean = (x0, p) match {
  case (seta(xs), p) => Lista.list_ex[A](p, xs)
}
```

```
def Ball[A](x0: set[A], p: A => Boolean): Boolean = (x0, p) match {
  case (seta(xs), p) => Lista.list_all[A](p, xs)
}
def remove[A : HOL.equal](x: A, xa1: set[A]): set[A] = (x, xa1) match {
  case (x, coset(xs)) => coset[A](Lista.insert[A](x, xs))
  case (x, seta(xs)) => seta[A](Lista.removeAll[A](x, xs))
}
def minus_set[A : HOL.equal](a: set[A], x1: set[A]): set[A] = (a, x1) match {
  case (a, coset(xs)) =>
    seta[A](Lista.filter[A](((x: A) => member[A](x, a)), xs))
  case (a, seta(xs)) =>
    Lista.fold[A, set[A]](((aa: A) => (b: set[A]) => remove[A](aa, b)), xs, a)
}
} /* object Set */
```

D. State Transition, Attack Trees, and Validity Predicate

```
object MC {
trait state[A] {
  val ‘MC.state_transition‘: (A, A) => Boolean
}
def state_transition[A](a: A, b: A)(implicit A: state[A]): Boolean =
  A.‘MC.state_transition‘(a, b)
object state {
}
} /* object MC */

object AT {
abstract sealed class attree[A]
final case class BaseAttack[A](a: (Set.set[A], Set.set[A])) extends attree[A]
final case class AndAttack[A](a: List[attree[A]], b: (Set.set[A], Set.set[A]))
  extends attree[A]
final case class OrAttack[A](a: List[attree[A]], b: (Set.set[A], Set.set[A]))
  extends attree[A]
def attack[A : MC.state](x0: attree[A]): (Set.set[A], Set.set[A]) = x0 match {
  case BaseAttack(b) => b
  case AndAttack(as, s) => s
  case OrAttack(as, s) => s
}
def is_attack_tree[A : HOL.equal : MC.state](x0: attree[A]): Boolean = x0 match
  {
  case BaseAttack(s) =>
    Set.Ball[A](Product_Type.fst[Set.set[A], Set.set[A]](s),
                ((x: A) =>
                  Set.Bex[A](Product_Type.snd[Set.set[A], Set.set[A]](s),
                             ((a: A) => MC.state_transition[A](x, a)))))
  case AndAttack(as, s) =>
    (if (Lista.nulla[attree[A]](as))
      Set.less_eq_set[A](Product_Type.fst[Set.set[A], Set.set[A]](s),
                         Product_Type.snd[Set.set[A], Set.set[A]](s))
```

```
      else (if (Lista.nulla[attree[A]](Lista.tl[attree[A]](as)))
             is_attack_tree[A](Lista.hd[attree[A]](as)) &&
               Product_Type.equal_prod[Set.set[A],
Set.set[A]](attack[A](Lista.hd[attree[A]](as)), s)
             else is_attack_tree[A](Lista.hd[attree[A]](as)) &&
                    (HOL.eq[Set.set[A]](Product_Type.fst[Set.set[A],
                   Set.set[A]](attack[A](Lista.hd[attree[A]](as))),
 Product_Type.fst[Set.set[A], Set.set[A]](s)) &&
                      is_attack_tree[A](AndAttack[A](Lista.tl[attree[A]](as),
              (Product_Type.snd[Set.set[A],
                                Set.set[A]](attack[A](Lista.hd[attree[A]](as))),
                Product_Type.snd[Set.set[A], Set.set[A]](s)))))))
  case OrAttack(as, s) =>
    (if (Lista.nulla[attree[A]](as))
      Set.less_eq_set[A](Product_Type.fst[Set.set[A], Set.set[A]](s),
                         Product_Type.snd[Set.set[A], Set.set[A]](s))
      else (if (Lista.nulla[attree[A]](Lista.tl[attree[A]](as)))
             is_attack_tree[A](Lista.hd[attree[A]](as)) &&
               (Set.less_eq_set[A](Product_Type.fst[Set.set[A], Set.set[A]](s),
                                   Product_Type.fst[Set.set[A],
              Set.set[A]](attack[A](Lista.hd[attree[A]](as)))) &&
                 Set.less_eq_set[A](Product_Type.snd[Set.set[A],
              Set.set[A]](attack[A](Lista.hd[attree[A]](as))),
                                    Product_Type.snd[Set.set[A],
               Set.set[A]](s)))
             else is_attack_tree[A](Lista.hd[attree[A]](as)) &&
                    (Set.less_eq_set[A](Product_Type.fst[Set.set[A],
                   Set.set[A]](attack[A](Lista.hd[attree[A]](as))),
 Product_Type.fst[Set.set[A], Set.set[A]](s)) &&
                      (Set.less_eq_set[A](Product_Type.snd[Set.set[A],
                    Set.set[A]](attack[A](Lista.hd[attree[A]](as))),
   Product_Type.snd[Set.set[A], Set.set[A]](s)) &&
                       is_attack_tree[A](OrAttack[A](Lista.tl[attree[A]](as),
              (Set.minus_set[A](Product_Type.fst[Set.set[A], Set.set[A]](s),
                                Product_Type.fst[Set.set[A],
            Set.set[A]](attack[A](Lista.hd[attree[A]](as)))),
                 Product_Type.snd[Set.set[A], Set.set[A]](s)))))))
}
} /* object AT */
```

References

[1] F. Kammüller, Isabelle infrastructure framework with iot healthcare s&p application, available at https://github.com/flokam/IsabelleAT. (2018).

[2] F. Haftmann, Code generation from Isabelle/HOL theories, accessed: 15.7.2018, with contributions from L. Bulwahn, Tutorial as part of the Isabelle documentation. Available from `www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle/doc/codegen.pdf` (December 2016).