Assuring the Guardians

Jonathan Laurent[1], Alwyn Goodloe[2], and Lee Pike[3]

[1] École Normale Supérieure, Paris, France,
jonathan.laurent@ens.fr,
[2] NASA Langley Research Center, Hampton, Virginia, USA,
a.goodloe@nasa.gov,
[3] Galois, Inc., Portland, OR, USA
leepike@galois.com

Abstract. Ultra-critical systems are growing more complex, and future systems are likely to be autonomous and cannot be assured by traditional means. Runtime Verification (RV) can act as the last line of defense to protect the public safety, but only if the RV system itself is trusted. In this paper, we describe a model-checking framework for runtime monitors. This tool is integrated into the Copilot language and framework aimed at RV of ultra-critical hard real-time systems. In addition to describing its implementation, we illustrate its application on a number of examples ranging from very simple to the Boyer-Moore majority vote algorithm.

1 Introduction

Runtime Verification (RV), where monitors detect and respond to property violations at runtime, can help address several of the verification challenges facing ultra-critical systems [20, 24]. As RV matures it will be employed to verify increasingly complex properties such as checking complex stability properties of a control system or ensuring that a critical system is fault-tolerant. As RV is applied to more complex systems, the monitors themselves will become increasingly sophisticated and as prone to error as the system being monitored. Applying formal verification tools to the monitors to ensure they are correct can help safeguard that the last line of defense is actually effective.

The work reported here is part of a larger program aimed at creating a framework for *high assurance RV*. In order to be used in ultra-critical environments, high-assurance RV must:

1. Provide evidence for a safety case that the RV enforces safety guarantees.
2. Support verification that the specification of the monitors is correct.
3. Ensure that monitor code generated implements the specification of the monitor.

These guiding principles inform the continued development of the Copilot language and framework that is intended to be used in RV of ultra-critical systems [18,22]. Earlier work focused on verifying that the monitor synthesis *process*

is correct (Requirement 3 above) [21]. Here, the focus is on the second requirement for high-assurance RV - making sure the monitor specification is correct. Requirement 1, in the spirit of Rushby's proposal [24] is future work.

Contributions In this paper we describe the theory and implementation of a k-induction based model-checker [5, 25] for Copilot called `copilot-kind`. More precisely, `copilot-kind` is a model-checking *framework* for Copilot, with two existing backends: a lightweight implementation of k-induction using Yices [4] and a backend based on *Kind2*, implementing both k-induction and the IC3 algorithm [26].

After providing a brief introduction to Copilot in Section 2 and to Satisfiability Modulo Theories (SMT)-based k-induction in Section 3, we introduce `copilot-kind` in Section 4. Illustrative examples of `copilot-kind` are provided in Section 5, and implementation details are given in Section 6. The final two sections discuss related work and concluding remarks, respectively.

Copilot and `copilot-kind` are open-source (BSD3) and in current use at NASA.[4]

2 Copilot

Copilot is a domain specific language (DSL) embedded in the functional programming language Haskell [14] tailored to programming monitors for hard real-time, reactive systems. Given that Copilot is deeply embedded in Haskell, one must have a working knowledge of Haskell to effectively use Copilot. However, the benefit of an embedded DSL in Haskell is that the host-language serves as a type-safe, Turing-complete macro language, allowing arbitrary compile-time computation, while keeping the core DSL small.

Copilot is a *stream* based language where a stream is an infinite ordered sequence of values that must conform to the same type. All transformations of data in Copilot must be propagated through streams. In this respect, Copilot is similar to Lustre [2], but is specialized for RV. Copilot guarantees that specifications compile to constant-time and constant-space implementations to update stream states.

Copilot's expression language. In the following, we briefly and informally introduce Copilot's expression language. Copilot streams mimic both the syntax and semantics of Haskell lazy lists with the exception that operators are automatically promoted point-wise to the list level.

Two types of temporal operators are provided in Copilot, one for delaying streams and one for looking into the future of streams:

```
(++) :: [a] → Stream a → Stream a
drop :: Int → Stream a → Stream a
```

[4] https://github.com/Copilot-Language

Here `xs ++ s` prepends the list `xs` at the front of the stream `s`. The expression `drop k s` skips the first `k` values of the stream `s`, returning the remainder of the stream. For example, the Fibonacci sequence modulo 2^{32} can be written in Copilot as follows:

```
fib :: Stream Word32
fib = [1,1] ++ (fib + drop 1 fib)
```

The base types of Copilot over which streams are built include Booleans, signed and unsigned words of 8, 16, 32, and 64 bits, floats, and doubles. Type-safe casts in which overflow cannot occur are permitted.

Sampling. Copilot programs are meant to monitor arbitrary C programs. They do so by periodically *sampling* values in the program under observation. Currently, Copilot can be used to sample variables, arrays, and the return values of side-effect free functions—sampling arbitrary structures is future work. For a Copilot program compiled to C, symbols become in-scope when arbitrary C code is linked with the code generated by the Copilot compiler. Copilot provides the operator `extern` to introduce an external symbol to sample.

The following stream samples the C variable `e0` of type `uint8_t` to create each new stream index. If `e0` takes the values `2, 4, 6, 8,...` the stream `ext` has the values `1, 3, 7, 13,...`.

```
ext :: Stream Word8
ext = [1] ++ (ext + extern "e0" )
```

3 Background on SMT-based k-induction

The focus of our investigation has been on applying model checking to prove invariant properties of our monitors. We employ a technique known as k-induction [5, 25] for verifying inductive properties of infinite state systems. k-induction has the advantage that it is well suited to SMT based bounded model checking. This section profiles the basic concepts of the k-induction proof technique needed in the remainder of the paper. In practice, we use tools that implement enhancements of the basic procedure such as path compression [3] that help the process scale, but are beyond the focus of the paper.

Consider a state transition system (S, I, T), where S is a set of states, $I \subseteq S$ is the set of initial states and $T \subseteq S \times S$ is a transition relation over S. To show P holds in the transition system one must show that (1) the base case holds—that P holds in all states reachable from an initial state in k steps, and (2) the induction step holds—that if P holds in states $s_0, \ldots, s_{k-1}$ then it holds in state s_k. The k-induction principle is formally expressed in the following two entailments:

$$I(s_0) \wedge T(s_0, s_1) \wedge \cdots \wedge T(s_{k-1}, s_k) \models P(s_k)$$
$$P(s_0) \wedge \cdots \wedge P(s_{k-1}) \wedge T(s_0, s_1) \wedge \cdots \wedge T(s_{k-1}, s_k) \models P(s_k)$$

If one cannot show the property to be true, the property is strengthened by either extending the formula or progressively increasing the length of the reachable states considered.

Property P said to be a k-inductive property with respect to (S, I, T) if there exists some $k \in \mathbb{N}^{0<}$ such that P satisfies the k-induction principle. As k increases, weaker invariants may be proved. If P is a safety property that does not hold, then the first entailment will break for a finite k and a counterexample will be provided. The trick is to find an invariant that is tractable by the SMT solver yet weak enough to satisfy the desired property.

4 Copilot Prover Interface

The `copilot-kind` model-checker is an extensible interface to multiple provers used to verify safety properties of Copilot programs. Currently, two backends for `copilot-kind` have been implemented: the first is a homegrown prover we call "the light prover" built on top of Yices [4] and the second is the Kind2 model checker being developed at the University of Iowa [17].

To begin, we describe how safety properties are specified in Copilot. Using the "synchronous observer" approach [10], properties *about* Copilot programs are specified *within* Copilot itself. In particular, properties are encoded with standard Boolean streams and Copilot streams are sufficient to encode past-time linear temporal logic [12]. We bind a Boolean stream to a property name with the `prop` construct in the specification, where the specification has type `Spec`.

For instance, here is a straightforward specification declaring one property:

```
spec = prop "gt0" (x > 0)
  where
  x = [1] ++ (1 + x)
```

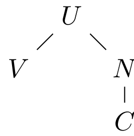
In order to check that property `gt0` holds, we use a `prove` function implemented as part of `copilot-kind`. Here, we can discharge the proof-obligation for the program above with the light prover using the command:

```
prove (lightProver def) (check "gt0") spec
```

where `lightProver def` stands for the *light prover* with default configuration.

While numeric types are bounded in Copilot, they are abstracted as integers in the prover, so we ignore overflow; see Section 6 for details.

Combining Provers. `Copilot-Kind` allows provers to be combined. Given provers A and B, the `combine` function returns a prover C which launches both A and B and returns the most *precise* output of the two upon termination. "Precise" in this case means returning the least element in the following partial order: for a given execution, classify prover outputs as valid (V), unknown (U), invalid with countexample (C), and invalid with no counterexample (N); the partial order is the least relation such that



(Merging provers that handle non-termination within a bound is future work.)

In practice, we used the following prover in the examples of Section 5:

```
prover = lightProver def {kTimeout = 5} ‘combine‘ kind2Prover def
```

which uses both the light and Kind2 provers, the first being limited to 5 steps of the k-induction.

Proof Schemes. Consider the example :

```
spec = do
  prop "gt0"  (x > 0)
  prop "neq0" (x /= 0)
  where
    x = [1] ++ (1 + x)
```

and suppose we want to prove `"neq0"`. Currently, the two available solvers fail at showing this non-inductive property (if at index i, `x` $= -k$, then it satisfies the induction hypothesis but fails the induction step for all k). Yet, we can prove the more general inductive lemma `"gt0"` and deduce our main goal from this. For this, we apply our proof scheme feature as follows:

```
assert "gt0"  >> check "neq0"
```

A *proof scheme* is a chain of primitive proof operations glued together by the `>>` operator to combine proofs, and in particular, provide lemmas. The available primitives are:

- `check "prop"` checks whether or not a given property is true in the current context.
- `assume "prop"` adds an assumption in the current context.
- `assert "prop"` is a shortcut for `check "prop" >> assume "prop"`.
- `assuming props scheme` assumes the list of properties *props*, executes the proof scheme *scheme* in this context, and forgets the assumptions.
- `msg "..."` displays a string in the standard output.

5 Examples

In this section, we will present several examples of `copilot-kind` applied to verify properties on Copilot monitors.

First, let us reexamine the Copilot program from Section 2 that generates the Fibonacci sequence. A fundamental property of this program is that it produces a stream of values that are always positive. We express this as follows:

```
spec = prop "pos" (fib > 0)
  where
    fib :: Stream Word64
    fib = [1, 1] ++ (fib + drop 1 fib)
```

This invariant property is clearly inductive and is easily discharged. Note that, as discussed in Section 6, 64-bit words are modelled by integers and eventual overflow problems are ignored here.

The next example uses `copilot-kind` to prove properties relating two different specifications. Consider the following specification:

```
intCounter :: Stream Bool → Stream Word64
intCounter reset = time
  where
    time = if reset then 0
           else [0] ++ if time == 3 then 0 else time + 1
```

that acts as a counter performing modulo arithmetic, but is reset when the `reset` stream value is true. Now consider the specification

```
greyTick :: Stream Bool → Stream Bool
greyTick reset = a && b
  where
    a = (not reset) && ([False] ++ not b)
    b = (not reset) && ([False] ++ a)
```

After a reset, `greyTick`'s output stream forms a cycle of Boolean values with the third item in the cycle having value `true` and the rest being `false`. Thus, the two specifications both have a cyclic structure and with a cycle that begins when the `reset` stream is set to `true`.

```
spec = do
  prop "iResetOk"   (r ==> (ic == 0))
  prop "eqCounters" (it == gt)
  where
    ic = intCounter r
    it = ic == 2
    gt = greyTick r
    r  = extern "reset" Nothing
```

Fig. 1. `Spec` listing.

From the above observations we conjecture that given the same input stream, when `reset` is true, the `intCounter` is 0 and `greyTick` is `true` when `intCounter` is 2. (Extern streams are uninterpreted; see Section 6.) We formalize these two

properties in our framework as shown in Figure 1. These predicates are discharged using the proof scheme

```
scheme :: ProofScheme
scheme = do
  check "iResetOk"
  check "eqCounters"
```

5.1 Boyer-Moore Majority Vote

Earlier research on Copilot has investigated fault-tolerant runtime verification [22]. Fault-tolerant algorithms often include a variant of a majority vote over values (e.g., sensor values, clock values, etc.). The Boyer-Moore Majority Vote Algorithm is a linear-time voting algorithm [13, 16]. In this case-study, we verify the algorithm.

The algorithm operates in two passes, first it chooses a candidate and the second pass verifies that the candidate is indeed a majority. The algorithm is subtle and the desire to apply formal verification to our Copilot implementation helped motivate the effort described here.

Two versions of this algorithm were checked with `copilot-kind`. The first algorithm was the one implemented as part of the aforementioned research on fault tolerance and flew on a small unmanned aircraft. This algorithm is a parallel implementation, where at each tick, the algorithm takes n inputs from n distinct streams and is fully executed. The second version of the algorithm is a sequential version, where the inputs are delivered one by one in time and where the result is updated at each clock tick. Both can be checked with the basic k-induction algorithm, but the proofs involved are different.

The parallel version. The core of the algorithm is the following:

```
majorityVote :: (Typed a, Eq a) => [Stream a] -> Stream a
majorityVote [] = error "empty list"
majorityVote (x : xs) = aux x 1 xs
  where
  aux p _s [] = p
  aux p s (l : ls) =
    local (if s == 0 then l else p) $ \ p' ->
    local (if s == 0 || l == p then s + 1 else s - 1) $ \ s' ->
    aux p' s' ls
```

Let us denote A as the set of the elements that can be used as inputs for the algorithm. Assume l is a list and $a \in A$, we denote $|l|_a$ as the number of occurrences of a in l. The total length of a list l is simply written $|l|$. The `majorityVote` function takes a list of streams l as its input and returns an output maj such that:

$$\forall a \in A,\ (a \neq maj) \Longrightarrow (|l|_a \leq |l|/2)$$

Given that quantifiers are handled poorly by SMT solvers and their use is restricted in most model-checking tools, including `copilot-kind`, we use a simple trick to write and check this property. If $P(n)$ is a predicate of an integer n, we have $\forall n \,.\, P(n)$ if and only if $\neg P(n)$ is unsatisfiable, where n an unconstrained integer, which can be solved by a SMT solver. The corresponding Copilot specification can be written as:

```
okWith :: (Typed a, Eq a) ⇒
          Stream a → [Stream a] → Stream a → Stream Bool
okWith a l maj = (a /= maj) ⟹ ((2 ∗ count a l) ≤ length l)
  where
    count _e [] = 0
    count e (x : xs) = (if x == e then 1 else 0) + count e xs

spec = prop "OK" (okWith (arbitraryCst "n") ss maj)
  where
    ss = [ arbitrary ("s" ++ show i) | i ← [1..10] ]
    maj = majorityVote
```

The function `arbitrary` is provided by the `copilot-kind` standard library and introduces an arbitrary stream. In the same way, `arbitraryCst` introduces a stream taking an unconstrained but constant value.

Note that we prove the algorithm for a fixed number of N inputs (here $N = 10$). Therefore, no induction is needed for the proof and the invariant of the Boyer-Moore algorithm does not need to be made explicit. However, the size of the problem discharged to the SMT solver grows in proportion to N.

The serial version. Now, we discuss an implementation of the algorithm where the inputs are read one by one in a single stream and the result is updated at each clock tick. As the number of inputs of the algorithm is not bounded anymore, a proof by induction is necessary and the invariant of the Boyer-Moore algorithm, being non-trivial, has to be stated explicitly. As stated in Hesselink [13], this invariant is:

$$\forall m \in A, \quad (m \neq p) \Longrightarrow (s + 2|l|_m \;\leq\; |l|) \;\;\wedge\;\; (m = p) \Longrightarrow (2|l|_m \;\leq\; s + |l|)$$

where l is the list of processed inputs, p is the intermediary result and s is an internal state of the algorithm. The problem here is that the induction invariant needs universal quantification to be expressed. Unfortunately, this quantifier cannot be removed by a similar trick like the one seen previously. Indeed, when an invariant is of the form $\forall x . P(x, s)$, s denoting the current state of the world, the induction formula we have to prove is:

$$\forall x . P(x, s) \,\wedge\, T(s, s') \models \forall x . P(x, s')$$

Sometimes, the stronger entailment

$$P(x, s) \,\wedge\, T(s, s') \models P(x, s')$$

holds and the problem becomes tractable for the SMT solver by replacing a universally quantified variable by an unconstrained one. In our current example, it is not the case.

Our solution to the problem of dealing with quantifiers is restricted to the case where A is finite and we replace each formula of the form $\forall x \in A\ P(x)$ by $\bigwedge_{x\in A} P(x)$. This can be done with the help of the `forAllCst` function provided by the `copilot-kind` standard library. It is defined as:

```
forAllCst ::(Typed a) ⇒
  [a] → (Stream a → Stream Bool) → Stream Bool
forAllCst l f = conj $ map (f ∘ constant) l
  where conj = foldl (&&) true
```

The code for the serial Boyer-Moore algorithm and its specification is then:

```
allowed :: [Word8]
allowed = [1, 2]

majority :: Stream Word8 → (Stream Word8, Stream Word8, Stream
    Bool)
majority l = (p, s, j)
  where
    p  = [0] ++ if s ≤ 0 then l else p
    s  = [0] ++ if p == l || s ≤ 0 then s + 1 else s - 1
    k  = [0] ++ (1 + k)

    count m = cnt
      where cnt = [0] ++ if l == m then cnt + 1 else cnt

    j = forAllCst allowed $ λ m →
          local (count m) $ λ cnt →
          let j0 = (m /= p) ⟹ ((s + 2 ∗ cnt) ≤ k)
              j1 = (m == p) ⟹ ((2 ∗ cnt) ≤ (s + k))
          in j0 && j1

spec = do
  prop "J" j
  prop "inRange" (existsCst allowed $ λ a → input == a)
  where
    input = externW8 "in" Nothing
    (p, s, j) = majority input

scheme = assuming ["inRange"] $ check "J"
```

We make the hypothesis that all the elements manipulated by the algorithm are in the set `allowed`, which is finite. The SMT proofs are generally exponential with respect to the number of variables, so this approach does not scale well.

6 Implementation

In this section, we shall outline the structure of the implementation of our Copilot verification system. After Copilot type-checking and compilation, a Copilot program is approximated so it can be expressed in a theory handled by most SMT solvers, as described below. Any information of no use for the model checking process is thrown away. The result of this process is encoded in *Cnub* format, which is is structurally close to the Copilot core format, but supports fewer datatypes and operators. Then, it can be translated into one of two available representation formats:

- The IL format: a list of quantifier-free equations over integer sequences, where each sequence roughly corresponds to a stream. This format is similar to the one developed by Hagen [8], but customized for Copilot. The *light prover* works with this format.
- The TransSys format: a modular representation of a *state transition system*. The *Kind2 prover* uses this format, which can be printed into Kind2's native format [17].

6.1 Approximating a specification

The complexity of the models that are built from Copilot specifications is limited by the power and expressiveness of the SMT solvers in use. For instance, most SMT solvers do not handle real functions like trigonometric functions. Moreover, bounded integer arithmetic is often to be approximated by standard integer arithmetic.

The Cnub format is aimed at approximating a Copilot specification in a format relying on a simple theory including basic integer arithmetic, real arithmetic, and uninterpreted functions. The stream structure is kept from the Copilot core format, but the following differences have to be emphasized:

- In contrast to the great diversity of numeric types available in Copilot, we restrain ourselves to three basic types which are handled by the SmtLib standard: `Bool`, `Integer`, and `Real`. Problems related to integer overflows and floating point arithmetic are ignored.
- Uninterpreted functions are used to model operators that are not handled. They are abstract as function symbols satisfying the equality:

$$(\forall i.\ x_i = y_i) \Longrightarrow f(x_1, \cdots, x_n) = f(y_1, \cdots, y_n).$$

  in the quantifier-free theory of uninterpreted function symbols, as provided by most SMT solvers.
- Copilot extern variables are modelled by unconstrained streams. Particular precautions have to be taken to model access to external arrays in order to express the constraint that several requests to the same index inside a clock period must yield the same result.

Excepting the first point, the approximations made are sound: they result in a superset of possible behaviors for the RV.

The problem of integer overflows can be tackled by adding automatically to the property being verified some bound-checking conditions for all integer variables. However, this solution can generate a great overhead for the proof engine. Moreover, it treats every program which causes an integer overflow as wrong, although this behaviour could be intended. An intermediate way to go would be to let the developer annotate the program so he can specify which bounds have to be checked automatically or to use the bit vector types of SmtLib, which will be implemented in a future release.

6.2 The Light prover and the IL format

Our homegrown prover relies on an intermediate representation format called IL. An IL specification mostly consists of a list of quantifier-free equations over integer sequences. These equations contain a free variable n which is implicitly universally quantified. The IL format is similar to the one used by Hagen [8].

A stream of type a is modeled by a function of type $\mathbb{N} \to a$. Each stream definition is translated into a list of constraints on such functions. For instance, the stream definition

```
fib = [1, 1] ++ (fib + drop 1 fib)
```

is translated into the IL chunk:

$$\begin{gathered} f : \mathbb{N} \to \mathbb{N} \\ f(0) = 1 \\ f(1) = 1 \\ f(n+2) = f(n+1) + f(n). \end{gathered}$$

Suppose we want to check the property `fib > 0` which translates into $f(n) > 0$. This can be done in two steps of the k-induction seen in Section 3 by taking

$$T[n] \;\equiv\; (\, f(0) = 1 \;\wedge\; f(1) = 1 \;\wedge\; f(n+2) = f(n+1) + f(n)\,)$$

$$P[n] \;\equiv\; (\, f(n) > 0\,)$$

and checking that both

$$T[0] \;\wedge\; T[1] \;\wedge\; \neg\,(P[0] \;\wedge\; P[1])$$

and

$$T[n] \;\wedge\; T[n+1] \;\wedge\; P[n] \;\wedge\; P[n+1] \;\wedge\; \neg P[n+2]$$

are non-satisfiable, the last one being equivalent to

$$(f(n+2) = f(n+1) + f(n)) \wedge (f(n) > 0) \wedge (f(n+1) > 0) \wedge (f(n+2) \leq 0) \wedge \cdots$$

This simple example illustrates that the construction of SmtLib requests from an IL specification is straightforward.

6.3 The Kind2 prover and the TransSys format

Recall that a state transition system is a triple (S, I, T), where S is a set of states, $I \subseteq S$ is the set of initial states, and $T \subseteq S \times S$ is a transition relation over S. Here, a state consists of the values of a finite set of variables, with types belonging to $\{\texttt{Int}, \texttt{Real}, \texttt{Bool}\}$. I is encoded by a logical formula whose free variables correspond to the state variables and that holds for a state q if and only if q is an initial state. Similarly, the transition relation is given by a formula T such that $T\,[\,q,\, q'\,]$ holds if and only if $q \to q'$.

The TransSys format is a modular encoding of such a state transition system. Related variables are grouped into *nodes*, each node providing a distinct namespace and expressing some constraints between its variables. A significant task of the translation process to TransSys is to flatten the Copilot specification so the value of all streams at time n only depends on the values of all the streams at time $n-1$ which is not the case in the Fibonacci example shown earlier. This is done by a simple program transformation which turns

```
fib = [1, 1] ++ (fib + drop 1 fib)
```

into

```
fib0 = [1] ++ fib1
fib1 = [1] ++ (fib1 + fib0)
```

After this, it is natural to associate a variable to each stream. Here, the variables `fib0` and `fib1` would be grouped into a single node in order to keep some structure in the representation of the transition system.[5] Such a modular transition system is almost ready to be translated into the Kind2 native format. However, we first have to merge each node's pair whose components are mutually dependent as Kind2 requires a topological order on its nodes.

7 Related Work

The research reported here builds on recent research conducted in a number of areas including formal verification, functional programming and DSL design, and RV.

Copilot has many features common to other RV frameworks aimed at monitoring distributed or real-time systems. There are few other instances of RV frameworks targeted to C code. One exception is Rmor, which generates constant-memory C monitors [11]. Rmor does not address real-time behavior or distributed system RV, though. To our knowledge no other RV framework has integrated monitor verification tools into their systems.

[5] Maintaining structure is important for two reasons. First, the model checker can use this structural information to optimize its search; see *structural abstraction* in [8]. Second, structured transition systems are easier to read, debug, and transform.

Haskell-based DSLs are of growing popularity and given that they are all embedded into the same programming language, they share many similarities with Copilot. For instance, Kansas Lava [7], which is designed for programming field programmable gate arrays, and Ivory [19], which is designed for writing secure autonomous systems, are both implemented using techniques similar to Copilot.

The ROSETTE extension to the Racket language [6] provides a framework for building DSLs that integrate SMT solvers. Smten [23] is a DSL with embedded SMT solvers that is targeted at writing satisfiability based searches.

As we have already mentioned, Copilot is similar in spirit to other languages with stream-based semantics, notably represented by the Lustre family of languages [15]. Copilot is a simpler language, particularly with respect to Lustre's clock calculus, focused on monitoring (as opposed to developing control systems). The work that is most relevant the research presented in this paper is the application of the Kind model checking tool to verify Lustre programs [9]. Kind and its most recent incarnation [17] is designed to model check Lustre programs and due to the similarities between Copilot and Lustre we targeted the Kind2 prover to be one of our back ends as well. Yet, to the best of our knowledge, the Boyer-Moore majority voting examples given in Section 5.1 are more sophisticated than published results using Kind with Lustre.

8 Conclusion

In this paper, we have presented the development of `copilot-kind` that enhances the Copilot RV framework with an integrated model-checking capability for verifying monitors and illustrated its applicability to verify a range of monitors.

In practice, our tool turned out to be very useful, indeed, even when the property being checked is not inductive or the induction step is too hard, it is very useful to test the first entailment of the k-induction algorithm for small values of k, proving the property cannot be violated in the first k time steps or displaying a counterexample trace. Many subtle bugs can be captured for reasonable values of k.

Yet, k-induction does have limitations. For instance, writing k-inductive specifications can be difficult. Newer advances like the IC3 algorithm, implemented by Kind2, are aimed at proving non-inductive properties by splitting it into concise and relevant inductive lemmas. However, our experiments showed that currently available tools fail at proving very simple properties as soon as basic arithmetic is involved.

The development of `copilot-kind` has reinforced the efficacy of the embedded DSL approach. Being embedded in a higher-order functional language facilitated the creation of a number of features such as our proof scheme capability. We have also found it quite advantageous to be able to write properties in the familiar style of Haskell programs. For instance, in Section 5.1, the function `forAllCst` for the serial Boyer-Moore example in that it uses both a `fold` and a

`map` operator to model finite conjunctions. Beyond our own purposes, we believe that other embedded DSL developers could use our designs in order to interface their languages with proof engines.

Having successfully applied our tool to rather sophisticated monitors, future extensions are planned. Given that we are focused on cyber-physical systems, the limitations of SMT-based provers go beyond the fact that they become prohibitively slow as the size of their input increases. SMT-solvers do not generally handle quantifiers or special real-valued functions well. A promising way to deal with both these issues would be an extension of the *proof scheme* system where properties involving arbitrary streams are seen as universally quantified lemmas which can be specialized and added to the proof context by an explicit use of a new `apply` directive. An interface to MetiTarski [1] will also allow us to automatically prove some of the mathematical properties of interest, but connecting to an interactive prover may also be necessary.

References

1. Behzad Akbarpour and L.C. Paulson. MetiTarski: an automatic theorem prover for real-valued special functions. *Journal of Automated Reasoning*, 44(3):175–205, 2010.
2. P. Caspi, D. Pialiud, N. Halbwachs, and J. Plaice. LUSTRE: a declarative language for programming synchronous systems. In *14th Symposium on Principles of Programming Languages*, pages 178–188, 1987.
3. Leonardo de Moura, Harald Rueß, and Maria Sorea. Bounded model checking and induction: From refutation to verification. In Andrei Voronkov, editor, *Computer-Aided Verification, CAV 2003*, volume 2725 of *Lecture Notes in Computer Science*, pages 14–26. Springer-Verlag, 2003.
4. Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, July 2014.
5. Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electr. Notes Theor. Comput. Sci.*, 89(4):543–560, 2003.
6. Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 530–541. ACM, 2014.
7. Andy Gill. Domain-specific languages and code synthesis using Haskell. *Commun. ACM*, 57(6):42–49, June 2014.
8. G. Hagen. *Verifying safety properties of Lustre programs: an SMT-based approach*. PhD thesis, University of Iowa, 2008.
9. G. Hagen and C. Tinelli. Scaling up the formal verification of lustre programs with smt-based techniques. In *Proceedings of the 8th International Conference on Formal Methods in Computer-Aided Design (FMCAD 08)*. IEEE, 2008.
10. N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*. Springer Verlag, June 1993.

11. Klaus Havelund. Runtime verification of C programs. In *Testing of Software and Communicating Systems (TestCom/FATES)*, pages 7–22. Springer, 2008.
12. Klaus Havelund and Grigore Roşu. Efficient monitoring of safety properties. *Int. J. Softw. Tools Technol. Transf.*, 6(2):158–173, 2004.
13. Wim H. Hesselink. The Boyer-Moore majority vote algorithm, 2005.
14. Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. http://haskell.org/, 2002.
15. Jan Mikáĉ and Paul Caspi. Formal system development with Lustre: Framework and example. Technical Report TR-2005-11, Verimag Technical Report, 2005.
16. Strother J. Moore and Robert S. Boyer. MJRTY - A Fast Majority Vote Algorithm. Technical Report 1981-32, Institute for Computing Science, University of Texas, February 1981.
17. University of Iowa: Kind Research Group. Kind 2: Multi-engine SMT-based Automatic Model Checker. http://kind2-mc.github.io/kind2/.
18. Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. Copilot: A hard real-time runtime monitor. In *Runtime Verification (RV)*, volume 6418, pages 345–359. Springer, 2010.
19. Lee Pike, Patrick C. Hickey, James Bielman, Trevor Elliott, Thomas DuBuisson, and John Launchbury. Programming languages for high-assurance autonomous vehicles: extended abstract. In *Programming Languages meets Program Verification*, pages 1–2. ACM, 2014.
20. Lee Pike, Sebastian Niller, and Nis Wegmann. Runtime verification for ultra-critical systems. In *Proceedings of the 2nd Intl. Conference on Runtime Verification*, LNCS. Springer, September 2011.
21. Lee Pike, Nis Wegmann, Sebastian Niller, and Alwyn Goodloe. Experience report: a do-it-yourself high-assurance compiler. In *Proceedings of the Intl. Conference on Functional Programming (ICFP)*. ACM, September 2012.
22. Lee Pike, Nis Wegmann, Sebastian Niller, and Alwyn Goodloe. Copilot: Monitoring embedded systems. *Innovations in Systems and Software Engineering*, 9(4), 2013.
23. Richard Uhler and Nirav Dave. Smten with satisfiability-based search. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 157–176. ACM, 2014.
24. John Rushby. Runtime certification. In *Eighth Workshop on Runtime Verification (RV08)*, volume 5289 of *LNCS*, pages 21–35, 2008.
25. M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, FMCAD '00, pages 108–125. Springer-Verlag, 2000.
26. F. Somenzi and A. R. Bradley. Ic3: Where monolithic and incremental meet. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, FMCAD '11, pages 3–8. FMCAD Inc, 2011.