Trustworthy Runtime Verification via Bisimulation (Experience Report)

RYAN G. SCOTT, Galois, Inc., United States
MIKE DODDS, Galois, Inc., United States
IVAN PEREZ, KBR @ NASA Ames Research Center, United States
ALWYN E. GOODLOE, NASA Langley Research Center, United States
ROBERT DOCKINS*, Amazon, United States

When runtime verification is used to monitor safety-critical systems, it is essential that monitoring code behaves correctly. The Copilot runtime verification framework pursues this goal by automatically generating C monitor programs from a high-level DSL embedded in Haskell. In safety-critical domains, every piece of deployed code must be accompanied by an assurance argument that is convincing to human auditors. However, it is difficult for auditors to determine with confidence that a compiled monitor cannot crash and implements the behavior required by the Copilot semantics.

In this paper we describe CopilotVerifier, which runs alongside the Copilot compiler, generating a proof of correctness for the compiled output. The proof establishes that a given Copilot monitor and its compiled form produce equivalent outputs on equivalent inputs, and that they either crash in identical circumstances or cannot crash. The proof takes the form of a bisimulation broken down into a set of verification conditions. We leverage two pieces of SMT-backed technology: the Crucible symbolic execution library for LLVM and the What4 solver interface library. Our results demonstrate that dramatically increased compiler assurance can be achieved at moderate cost by building on existing tools. This paves the way to our ultimate goal of generating formal assurance arguments that are convincing to human auditors.

CCS Concepts: • **Software and its engineering → Formal software verification**; **Software verification**; Embedded software.

Additional Key Words and Phrases: Runtime verification, formal methods, assurance

ACM Reference Format:
Ryan G. Scott, Mike Dodds, Ivan Perez, Alwyn E. Goodloe, and Robert Dockins. 2023. Trustworthy Runtime Verification via Bisimulation (Experience Report). *Proc. ACM Program. Lang.* 7, ICFP, Article 199 (August 2023), 17 pages. https://doi.org/10.1145/3607841

1 INTRODUCTION

Safety-critical cyber-physical systems (CPSs) are subject to strict regulation to ensure public safety. Historically, such systems are constructed using conservative requirements-driven practices, yielding predictable systems that are amenable to verification by testing [RTCA 2011; SAE International 2010]. There is increasingly a desire to use off-the-shelf components and employ techniques like

*Author's paper contributions were made while working at Galois, Inc.

Authors' addresses: Ryan G. Scott, Galois, Inc., United States, rscott@galois.com; Mike Dodds, Galois, Inc., United States, miked@galois.com; Ivan Perez, KBR @ NASA Ames Research Center, United States, ivan.perezdominguez@nasa.gov; Alwyn E. Goodloe, NASA Langley Research Center, United States, a.goodloe@nasa.gov; Robert Dockins, Amazon, United States, rdoc@amazon.com.

2475-1421/2023/8-ART199
https://doi.org/10.1145/3607841

machine learning to build autonomous systems, but these cannot be assured using traditional approaches [Cofer 2021; Cofer et al. 2020; Council 2014; Members 2020].

Runtime verification (RV) [Falcone et al. 2013; Goodloe and Pike 2010] addresses this problem by monitoring a system under observation and responding to property violations during the mission. For example, an RV system might monitor engine heat levels, aircraft location within an authorized airspace, or autopilot changes between flight modes. While not static formal verification, RV provides a significant improvement in assurance for systems over testing alone.

Copilot [Perez et al. 2020; Pike et al. 2010, 2013] is a language and toolchain for writing RV monitors. Monitors are written in a high-level, stream-based domain-specific language (DSL) that is embedded in Haskell. Copilot is equipped with a compiler that generates C code, which can then be linked against other application code for use in production.

When used in safety-critical applications, Copilot monitors also form part of the safety case that justifies the system's mission readiness. As a result, monitors must be trustworthy, and trust must be based on systematic and rigorous evidence that can be audited. Specifically, Copilot monitors must be trustworthy in two regards. First, the monitor must be *validated*, meaning that the monitor enforces the higher-level properties that were intended. Second, the C implementation of the monitor must be *verified*, meaning that the compiled code correctly realizes the expected semantics of the monitor. The problem of *validation* is about establishing properties of the monitor program itself, and the problem of *verification* is about the correctness of the Copilot compiler. A number of mechanisms can be used to validate and verify the monitors. In recent years, formal methods have been increasingly accepted as a means of evidence in domains such as civil aerospace. For example, DO-333 [RTCA 2011] gives official guidance on the use of formal methods in certification practice.

This paper presents CopilotVerifier, which addresses the problem of verification.[1] More precisely, a monitor in the Copilot DSL has an executable semantics and, once compiled, the resulting C program also has an executable semantics. CopilotVerifier proves that the monitor and the C program are *extensionally equal*: given a stream of inputs, the monitor and its compiled form produce equivalent outputs. We also prove that either (1) both are memory safe, i.e. they cannot crash, or (2) that they crash exactly on equivalent inputs (the tool supports both modes).

CopilotVerifier takes a *translation validation* [Pnueli et al. 1998] approach rather than verifying the compiler for all possible programs. That is, it runs alongside the Copilot compiler and generates a proof for the particular compiled output. The proof establishes a bisimulation between the Copilot program and the compiled result. At each program point, it establishes that the state of the Copilot monitor corresponds to the state of the C program. Our verifier is designed to be push-button: given a Copilot monitor as input, the verifier automates the steps needed to construct a proof.

This problem of compiler correctness is familiar from research projects such as CompCert [Leroy 2009] and CakeML [Kumar et al. 2014], but these have tended to be multi-year multi-person efforts. This level of effort was not available to us as an industry project operating under budget constraints. Our objective was more pragmatic: to maximize our confidence in the Copilot compiler using off-the-shelf formal methods tools and libraries. We make several design choices that reduce the cost of building CopilotVerifier but, in some cases, reduce the power of our results (see Section 6). We judge these to be low-cost tradeoffs relative to the large increase in confidence we have achieved by successfully completing the CopilotVerifier project in just under one year. Overall, our results have been heartening: through careful design, CopilotVerifier shows that it is possible to apply formal assurance to a DSL compiler with modest levels of effort.

Our ultimate goal, of which this project is the first step, is to automatically generate formal evidence that is convincing to human auditors. By providing a proof of equivalence between the

[1]For validation, Copilot supports reasoning about monitor specs using theorem provers [Goodloe 2016; Laurent et al. 2015].

monitor and the specification as supporting evidence, we aim to build a certification argument based in formal methods rather than software engineering practices.

The rest of the paper is structured as follows. In Section 2, we describe the Copilot language and illustrate the problem that CopilotVerifier addresses. In Section 3, we give a high-level description of CopilotVerifier, and we illustrate our work by showing bugs in Copilot that CopilotVerifier helped identify. Section 4 discusses the bisimulation relation between Copilot and C programs. Section 5 presents an example of evidence that the verifier produces for assurance cases. In Section 6 we elaborate on the design trade-offs made in CopilotVerifier. We close the paper with a discussion on related work (Section 7) and conclusions (Section 8).

2 COPILOT OVERVIEW

Copilot is an embedded domain-specific language (EDSL) implemented as a Haskell library. This section provides a brief overview of the Copilot language and the aspects of its semantics relevant to our verification task. Readers interested in further details should consult [Perez et al. 2020]. Readers familiar with Copilot can skip to Section 2.4 for a discussion of the challenge of verifying Copilot compilation. Note that we generally refer to programs written in the Copilot DSL as *monitors*, but they are also often called *specs* or *specifications*, reflecting their role in runtime verification.

2.1 Copilot Streams

The main programming abstraction in Copilot is the *stream*, which represents a discrete sequence of values over time. Values include integers, floating-point numbers, booleans, and compound types like structs and arrays. Copilot programs are evaluated one step at a time, either at regular intervals or as new data becomes available. The time between each value in a stream is abstract and application-dependent; it can be thought of as a constant corresponding to a unit of real time.

To interact with the system being monitored, Copilot programs may reference *external* streams, which generally represent values obtained from the system being monitored (e.g., sensor data). From these external streams, other intermediate streams of data are computed, eventually resulting in boolean streams that capture the conditions being monitored. At execution time, an external handler function is called whenever a *trigger stream* evaluates to true, which allows the execution environment to take action.

Figure 1 shows a complete Copilot program implementing a simple thermostat. The program monitors an external stream (`temp`) representing data from a temperature probe, firing a `trigger` whenever the temperature (`avgTemp`) is sufficiently above or below a fixed setpoint, each captured by a boolean stream in the `spec`. To provide some robustness against noisy data, the probe input data is smoothed by computing a sliding window average of the last 5 samples.

This program demonstrates how to construct streams of constant values using `constant`, as well as how to cast from a stream of `Word8`s to a stream of `Float`s using `unsafeCast`.[2] The thermostat program also demonstrates some operations on streams whose implementations differ from the Haskell functions of the same name. Operations such as multiplication `(*)`, subtraction `(-)`, and comparison `(>)` work pointwise over the elements of a stream. The `(++)` operator prepends a list containing a fixed number of samples to the front of a stream. Invoking `sum n s` will compute a stream where the ith element of the stream consists of the sum of the elements in `s` from indices i through $i + (\texttt{n} - 1)$.

2.2 Reifying Copilot Specifications

The first step when working with any Copilot monitor (whether to simulate, prove properties, or generate code) is to *reify* the monitor into an intermediate representation called Copilot Core.

[2]This cast is safe, since there is no information loss in this direction. Casting in the other direction could lose information.

```
-- External temperature as a
-- byte, from -50C to 100C
temp :: Stream Word8
temp = extern "temperature" Nothing

-- Calculate temperature in Celsius
ctemp :: Stream Float
ctemp =
  (unsafeCast temp*constant (150.0/255.0))
  - constant 50.0

-- Width of the sliding window
window :: Int
window = 5

-- Compute sliding average of last 5 temps
avgTemp :: Stream Float
avgTemp =
  (sum window
       (replicate window 19.5 ++ ctemp))
  / fromIntegral window

spec :: Spec
spec = do
  trigger "heaton" (avgTemp < 18.0)
    [arg avgTemp]
  trigger "heatoff" (avgTemp > 21.0)
    [arg avgTemp]
```

Fig. 1. A simple thermostat example.

```
(Float) s0 = [19.5,19.5,19.5,19.5,19.5] ++ (((cast) Ext_temperature * 0.5882353) - 50.0)
trigger "heaton" =
   ((((((s0 + drop 1 s0) + drop 2 s0) + drop 3 s0) + drop 4 s0) / 5.0) < 18.0)
   [arg (((((s0 + drop 1 s0) + drop 2 s0) + drop 3 s0) + drop 4 s0) / 5.0)]
trigger "heatoff" =
   ((((((s0 + drop 1 s0) + drop 2 s0) + drop 3 s0) + drop 4 s0) / 5.0) > 21.0)
   [arg (((((s0 + drop 1 s0) + drop 2 s0) + drop 3 s0) + drop 4 s0) / 5.0)]
```

Fig. 2. The reified version of `spec` in Figure 1.

In addition to unfolding helper definitions to produce the intermediate syntax, the reification step checks that the streams verified are *well-formed*. Well-formedness is a syntactic check that ensures that no computations violate temporal causality (i.e., they only depend on "past" values) and that they require only a finite history to compute. These properties ensure that monitors can be implemented using a step-by-step strategy that requires only a constant amount of memory.

The reified version of the thermostat example is shown in Figure 2. Here, it becomes clear how the sliding window calculation is unfolded into a stream computation involving the `drop` operator to sum up the 5 most recent values of the stream named `s0`. Given a number `n`, `drop n` drops the first `n` elements from a stream, where the stream must have at least `n` elements of history available. As a result, the `s0` stream must have a history of at least 5 values, and we must initialize it with an array of constant values. The `cast` operator is the reified counterpart to `unsafeCast` in the stream language.

2.3 Compiling to C Code

The CopilotC99 library takes a reified monitor and generates C code. The generated code is not a standalone program and is intended to be used part of an existing control system. To facilitate integration, the code is free of external dependencies and targets the C99 subset of the C language.

There is a straightforward translation from each possible stream value to a C value. For instance, an `Int32` in a stream program would be translated to an `int32_t` in the generated C code, and similarly for other scalar types. Compound stream values such as structs and arrays are translated to C structs and arrays with corresponding struct field names and array lengths.

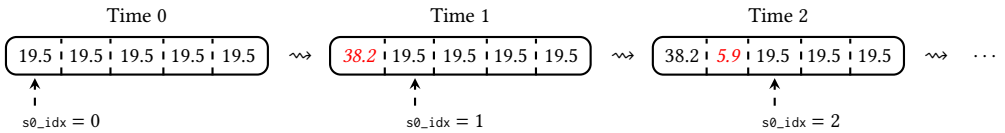
While Copilot streams are conceptually infinite sequences, Copilot-generated C programs only use a finite amount of memory. Each stream is translated to a ring buffer, implemented as an array with fixed length equal to the history needed to compute the stream. Each buffer has an associated index that tracks the current position in the array, which is incremented as time advances.

```
extern uint8_t temperature;
void heaton(float arg0);
void heatoff(float arg0);

static float s0[5] = {19.5f, 19.5f, 19.5f,
                      19.5f, 19.5f};
static size_t s0_idx = 0;

float s0_get(size_t x) {
  return s0[(s0_idx + x) % 5];
}

float avgTemp(void) {
 return (s0_get(0) + s0_get(1) + s0_get(2)
         + s0_get(3) + s0_get(4)) / 5.0f;
}
void step(void) {
  if (avgTemp() < 18.0f) heaton(avgTemp());
  if (avgTemp() > 21.0f) heatoff(avgTemp());
  s0[s0_idx] =
    (temperature * 0.5882353f) - 50.0f;
  s0_idx = (s0_idx + 1) % 5;
}
```

Fig. 3. C code generated for Figure 1. The code has been cleaned up for presentation purposes only.

Figure 3 shows the generated C code for the thermostat example. The `s0` stream is translated to C as a ring buffer with 5 elements, just large enough to store the initial 5 temperature values. This ring buffer will be used to compute the current values for the various streams appearing in the trigger definitions, and it will be updated based on the current value of the external `temperature` stream on each tick. A more complicated monitor may have additional ring buffers of various lengths, each of which will be updated as necessary.

The `step` function advances the state of the program by a single tick. The following diagram shows how `step` computes subsequent temperatures from the previous time step and inserts into the appropriate position in the buffer, tracked by the index value `s0_idx`. This diagram assumes that the inputs from the external `temperature` stream begin [150, 95, ...]. One additional temperature value is computed at each time step, which is marked in this diagram in red italic font.



Aside from updating the ring buffer, the `step` function is responsible for checking if any of the monitored conditions have been met and, if so, firing the corresponding triggers. The trigger functions `heaton` and `heatoff` are only given forward declarations; they are meant to be defined by the application that links against the Copilot monitor.

2.4 The Challenge of Verifying Copilot Compilation

We have seen how the thermostat example has been translated from a stream-based program (Figure 1) to a C program (Figure 3), but can we be sure that they actually behave in identical ways? With careful evaluation, one can see that the values in the `s0` buffer, as computed by `s0_get(0)`, `s0_get(1)`, ..., and `s0_get(4)` at a given time step n, will match the nth, $(n+1)$th, ..., and $(n+4)$th elements of the `avgTemp` stream. As a result, the `heaton` and `heatoff` trigger functions in C will be called if and only if the corresponding trigger streams evaluate to true, and the arguments to the trigger functions will always equal the value of the `avgTemp` stream at that time step. In this sense, the two programs exhibit the same behavior. Most Copilot monitors in the wild are significantly more involved than this example, however, and establishing a correspondence between the monitor and the generated C code is substantially harder.

3 VERIFIER OVERVIEW

CopilotVerifier runs alongside the CopilotC99 compiler and formally verifies that the results are correct. It does this by proving that the semantics of the input Copilot monitor correspond to the semantics of the generated C program. In most cases, the verifier requires no input annotations

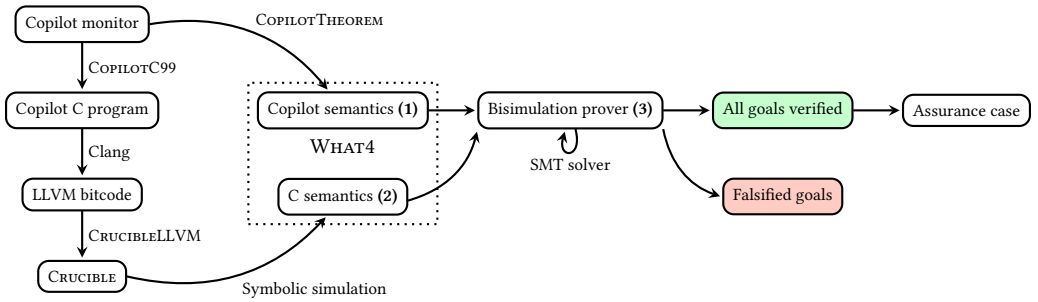


Fig. 4. The architecture of CopilotVerifier.

beyond what is supplied to the compiler (we discuss a few exceptions in Section 6.3). The full implementation of the verifier can be found in our supplementary artifact [Scott et al. 2023].

The architecture of CopilotVerifier is depicted in Figure 4. At its heart is a comparison between the input Copilot program, which computes over streams **(1)**, and the compiled C program, which represents streams in real memory **(2)**. To enable this comparison, both the input and compiled program are represented by a collection of formulas in What4, an SMT interface library [Hendrix et al. 2020]. The relationship between the two semantics are then established by the bisimulation prover **(3)**. CopilotVerifier reports back to the user whether it solved all of the generated proof goals successfully. If it succeeds, the verifier can produce an assurance argument that the proof is valid, as explained in Section 5. If not, the verifier identifies which goals were falsified.

Intuitively, programs at both the Copilot and C levels are translated to transition systems corresponding to the program control-flow graph. The formulas that are generated in What4 encode the effect of a single transition on the state—either streams or memory. The generated transition systems are intentionally very similar in structure, so the main task of the bisimulation proof is to demonstrate that the states stay in correspondence (see Section 4).

The translation from Copilot programs to What4 is performed by CopilotTheorem, which we developed. Because Copilot is a functional language, it is relatively simple to encode the semantics of each program step in an SMT style. For instance, Copilot's integer arithmetic operations involving streams translate straightforwardly to SMT-Lib's fixed-size bitvector operations, so CopilotTheorem would translate the expression `x + 42 :: Stream Int32` into an SMT formula resembling `(bvadd x (_ bv42 32))`. There are similarly direct translations for all other stream operations with the exception of floating-point operations, a special case that we discuss further in Section 6.3.1.

It is much more complex to faithfully capture the semantics of a C program. To do this, we lean on a pre-existing tool, Galois's Crucible [Christiansen et al. 2019] symbolic simulation library. Specifically, CopilotVerifier compiles the C program to LLVM bitcode and uses Crucible's LLVM backend to simulate it. The result is a collection of What4 formulas that precisely represent the LLVM program's semantics. Crucible provides an accurate model of LLVM, intended for industry formal methods applications. For example, Crucible has previously been applied in tools used to verify industry cryptographic libraries [Boston et al. 2021; Chudnov et al. 2018; Dockins et al. 2016].

Bugs Found in Copilot. We developed CopilotVerifier by testing it against a variety of examples used in Copilot's test suite, which uncovered a number of bugs in Copilot itself. These issues include memory unsafety [GitHub 2021e, 2022c], incorrect C code generation [GitHub 2021a,d, 2022a,b], and mismatches between Copilot and C semantics [GitHub 2021b,c]. All of these issues have since been fixed upstream.

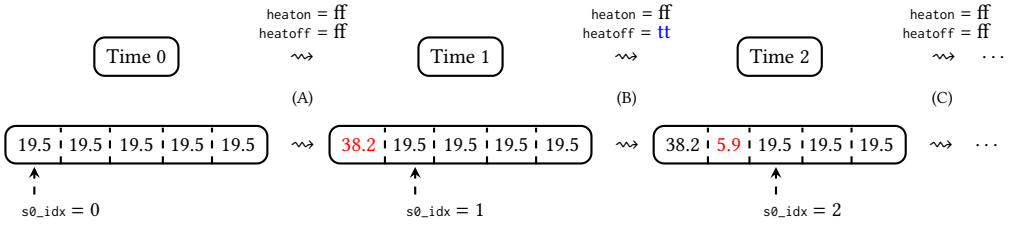


Fig. 5. Two LTSes. The upper LTS represents the stream program in Figure 1. The lower LTS represents the C program in Figure 3. Transition labels are shared between both LTSes. Transition B fires the `heatoff` trigger.

4 BISIMULATION PROOF STRUCTURE

The core theorem we wish to prove is *extensional equality*. That is, the Copilot program and its compiled C representation behave identically in the input-output behavior that can be observed by the execution context. Only trigger functions can be observed in Copilot, which leads us to the following more formal description. A copilot program P and its compiled form Q are extensionally equal if for any arbitrary input stream, the following holds at every time step:

- The same set of trigger functions are called in P and Q with the same arguments.
- P has crashed iff C has crashed.

Proving extensional equality between arbitrary programs is difficult, but the Copilot program and the compiled program are intentionally very similar in their structure. Consider again the Copilot thermostat program in Figure 1 and the resulting C code in Figure 3 (we will use this as a running example in this section). Let us assume that the C program's first n inputs correspond to the first n values of the external stream inputs. Intuitively, after n calls to the `step` function in the C program, the state of the ring buffers should be equal to the value of the corresponding stream expressions at index n. Moreover, the trigger functions in the C program should be called from the `step` function at the same times when the corresponding stream expressions evaluate to true.

We can view a Copilot stream program and its generated C program as labeled transition systems (LTSes). To prove correctness, CopilotVerifier constructs a bisimulation relation between the two systems. Intuitively, the proof shows that the two systems start in corresponding states, and every transition in one system has a transition to a corresponding state in the other system. This has the effect of proving extensional equality because it shows that trigger functions are called in corresponding states, and that the two systems transition to crashing states at the same time.

To be precise, CopilotVerifier does not prove the final bisimulation property. Rather, it proves a set of per-transition properties whose conjunction implies a bisimulation. Verifying the bisimulation would take us outside the logical fragment which can be easily reasoned about in SMT solvers.

4.1 Programs as Transition Systems

Formally, an LTS consists of a set of states, a set of labels, and a set of labeled transitions between pairs of states. Consider again the thermostat example (Figures 1 and 3). Let us assume that the inputs from the external `temperature` stream begin $[150, 95, \ldots]$. In Celsius, these roughly correspond to 38.2° and 5.9°. Note that the triggers `heaton` and `heatoff` will fire if the sliding average of the previous five temperatures dips below 18° or if it exceeds 21°, respectively.

Copilot is a stream processing language based on functional programming ideas. As a result, it has no explicit state in its semantics, beyond the (immutable) input streams and the current time step. Instead, the value of stream expressions are calculated from the input stream at the current time step and its prefix. A sequence of transitions for the thermostat stream program is pictured in the upper diagram of Figure 5.

In a C program, each state consists of the global memory that contains the ring buffers and their current indices. The transition relation is defined by the generated `step` function. An LTS for the thermostat C program is pictured in the lower diagram of Figure 5. This program's global memory only tracks one buffer, `s0`, which holds the five most recent temperatures. The global memory also tracks `s0_idx`, the current index into the buffer. In each transition, a newly sampled value from `temperature` is placed in the position that `s0_idx` points to, after which the index is incremented.

The same set of labels is used for both the stream and C LTSes. Each label is associated with observable events required for the transition to occur. In the stream spec, the events correspond to trigger streams evaluating to true. In C, the events correspond to the `step` function invoking the corresponding trigger functions.

In Figure 5, each transition records whether the `heaton` and `heatoff` triggers were fired. The `heatoff` trigger fires in transition (B), as adding 38.2° makes the sliding average temperature 23.2°, which exceeds the upper bounds that `avgTemp` checks for. In transition (C), however, `heatoff` no longer fires, as adding 5.9° lowers the average temperature to 20.5°.

4.2 Correspondence Relation

We now define a *correspondence relation* between stream and C program states. Copilot is designed so that each stream always has a finite window of past values that can be accessed at any point in the program. Consider a Copilot program containing a stream s with a window value k. Let `buf` be the ring buffer in the C program that corresponds to s, and let `idx` be the current index into `buf`. A stream program state is related to a C program state by the correspondence relation if and only if the value of s at index $n + i$ is equal to the value of `buf` at index $(\texttt{idx} + i) \bmod k$, where i ranges from 0 to $k - 1$. We lift this to sets of streams in the obvious way.

The thermostat example has a single stream definition `s0` that retains `window = 5` previous values. At time step 0, the `s0` stream's first five elements are `19.5` due to the use of `replicate window 19.5` in its definition. In C, the values of `s0_get(i)` (as `i` ranges from `0` to `4`) are also `19.5`, as these are the initial values of the `s0` buffer before running `step`. Therefore, the two programs are in correspondence at time step 0. We can also intuit that the two programs correspond at subsequent time steps. For instance, at time step 1, the `s0` stream would produce `38.2` as its fifth value, which would match the value of `s0_get(4)` after a single invocation of the `step` function on the C side.

4.3 Proving the Bisimulation

The goal of CopilotVerifier is to demonstrate that the correspondence relation is a bisimulation. That is, for every pair of states (s, c) in the correspondence relation, if s transitions to s' in the stream program with label α, then there must exist a C program state c' such that c transitions to c' with label α. The converse must also be true: if c transitions to c', then there must exist a stream program state s' such that s transitions to s' with label α.

It is straightforward to identify which transitions in each LTS correspond to each other, as the nth time step in the stream program corresponds to the nth invocation of `step` in the C program. The main challenge, then, is to demonstrate that the stream values equal the ring buffer values at each time step. The verifier does this by proving three properties about the correspondence relation:

(1) The initial states of the programs are in the correspondence relation.
(2) For each pair of states (s, c) in the correspondence relation, there exist a stream program state s', a C program state c', and a label α such that s transitions to s' with label α, c transitions to c' with label α, and (s', c') is in the correspondence relation.
(3) For each label α used in the transition relations, the triggers for α fire in corresponding ways in both programs.

Note that the definition of a bisimulation has two directions: one direction in which the stream program state s' is universally quantified and the C program state c' is existentially quantified, and another direction in which the order of quantification is reversed. Property (2), on the other hand, checks both of these directions simultaneously. This is done for practical reasons, as it is always clear what the existentially quantified states s' and c' will be after each transition. As such, the verifier combines both of these directions into a single step.

CopilotVerifier reduces each of these properties to SMT formulas. The proof principle of bisimulation itself is not amenable to SMT, as it falls outside of the first-order theories that SMT solvers understand. Likewise, the semantics of Copilot and C could potentially be reduced to SMT, but it would be impractical to do so. Instead, we reduce the individual proof obligations listed above into a series of lower-level logical statements that can be represented with SMT queries.

4.3.1 Initial State Correspondence. The first proof obligation that the verifier must discharge is that the initial states of the two programs correspond. This is tantamount to taking the initial values in each ring buffer and proving that they are equal to the first k values of the corresponding stream at time step 0, where k is the length of the ring buffer. Due to the restrictions that Copilot places on programs, these first k values must be concrete and cannot depend on external inputs. As a result, this step is simple to translate to SMT queries and only requires evaluation of concrete values.

Thermostat example: We demonstrated initial state correspondence for the thermostat example in Section 4.2. The proof is tantamount to showing that the values of the `s0` stream and ring buffer all equal `19.5` at time step 0, a total of five proof goals.

4.3.2 Transition Correspondence. Most of the proof effort consists of demonstrating that the correspondence relation is preserved by transitions. In this phase of the proof, we must begin with completely symbolic program states at an arbitrary time value n. As a result, we must create fresh symbolic values for each stream definition and its corresponding ring buffer.

More precisely, let s range over the stream definitions, where each s is required to retain k previous values. Let `buf` be the ring buffer in the C program that corresponds to s, and let `idx` be the current index into `buf`. For each i ranging from 0 to $k - 1$, we create a symbolic value and assume it is equal to both the value of s at index $n + i$ and the value of `buf` at index (`idx` + i) mod k. We then advance the symbolic state of the stream and C programs once. Then, for each j ranging from 1 to k, we read the value of s at index $n + j$ and check that it is equal to the value of `buf` at index (`idx` + j) mod k under the previous assumptions.

Advancing the symbolic state of the stream program is a matter of evaluating each stream expression at the next time step. For the C program, we advance the symbolic state by invoking the Crucible symbolic simulator to run the `step` function, which updates the memory used in the program. In addition to generating proof goals about bisimulation equivalence conditions, Crucible can also generate side conditions that relate to the memory safety of the program. For instance, each memory access into a ring buffer could potentially have out-of-bounds indexing. If CopilotC99 generates C code correctly, all such accesses should be within bounds, but this must be checked as a part of the query submitted to the SMT solver. The simulator also generates side conditions related to C operations with undefined behavior—see Section 4.3.4 for details.

Thermostat example: Ten goals involve checking symbolic values for equality, with two goals discharged for each element that the `s0` stream retains. Thirty goals involve checking if the array indexes in `s0_get` and `step` are within bounds. Four goals involve checking if the trigger functions fire in equivalent ways, which is described in Section 4.3.3. In total, the verifier discharges 44 goals.

4.3.3 Triggers. The proof must establish that, for each trigger function in the generated C program, the trigger function is called if and only if the corresponding trigger stream in the stream program

evaluates to true. In a real setting, the application linked against a Copilot monitor would implement the trigger functions. However, we are verifying the generated monitor code in isolation, so we do not have implementations of the trigger functions available. Instead, the verifier creates stub implementations for each trigger function that captures the arguments and the path condition under which it was called. After symbolic simulation finishes, the captured arguments and path condition are asserted to be equivalent to the corresponding trigger stream and arguments from the stream program.

Thermostat example: Four proof goals arise from checking for equivalent behavior for the `heaton` and `heatoff` triggers in the example. Two goals check whether each trigger is fired in both programs during a transition. Two goals check that the arguments passed to each trigger correspond.

4.3.4 Partial Operations. Another subtlety of the transition relation step of the proof is how to handle partial Copilot operations. These range from division, which can fail if the second argument is zero, to signed integer arithmetic, which can overflow. If a partial operation is used on an input for which it is not defined, it can result in undefined behavior in the generated C code.

One way to handle partial operations is to take an uncompromising approach: if Crucible detects undefined behavior when simulating the generated C code, it aborts and causes the proof to fail. This ensures that Copilot specifications do not have any misbehavior, but, if a spec *does* misbehave, the verifier will simply fail and not reveal anything about the rest of the spec.

Another way to handle partial operations is to prove that a Copilot spec is "crash-equivalent" to its corresponding C program. That is, if the C program invokes a partial operation on undefined inputs, the verifier will check that this corresponds to an invocation of the corresponding operation in the Copilot spec on the same inputs. CopilotVerifier supports both the uncompromising approach and the crash-equivalence approaches as user-configurable options.

To check for crash-equivalence during symbolic simulation, the verifier will analyze any invocation of an operation in the stream program which could be partial and generate a side condition that this operation will only be invoked on well defined inputs. During the transition step of the proof, the verifier will assume these side conditions before starting symbolic simulation. Therefore, if the simulator generates any side conditions due to partial operations in the C program, they should be dischargeable using the corresponding side conditions from the stream program.

5 ASSURANCE CASES

CopilotVerifier is the first step in a longer project: integrating formal methods into safety-critical deployment practices. Our ultimate goal is for Copilot code to be deployed with a safety case largely constructed by automated means. Existing standards such as DO-333 provide guidance on how formal methods evidence should be handled. However, there are few examples of successful formal methods deployments in safety-critical practice. See [Cofer and Miller 2014; Wagner et al. 2017] for a discussion of the issues involved in certification and formal methods.

To use Copilot in safety-critical systems, the evidence provided by our tools must be understood and accepted by human auditors. This is challenging because CopilotVerifier performs most of its reasoning through complicated SMT queries. These queries are difficult to analyze manually without some way of connecting these queries back to higher-level requirements.

We address this challenge by giving CopilotVerifier an optional setting that, when enabled, displays each Crucible proof goal that is generated during a successful run of the verifier. Each proof goal has an accompanying high-level description of what it has demonstrated, such as a symbolic stream value being equal to its corresponding C ring buffer value. Each proof goal also has an associated What4 formula and SMT query representing the goal. With this information, each

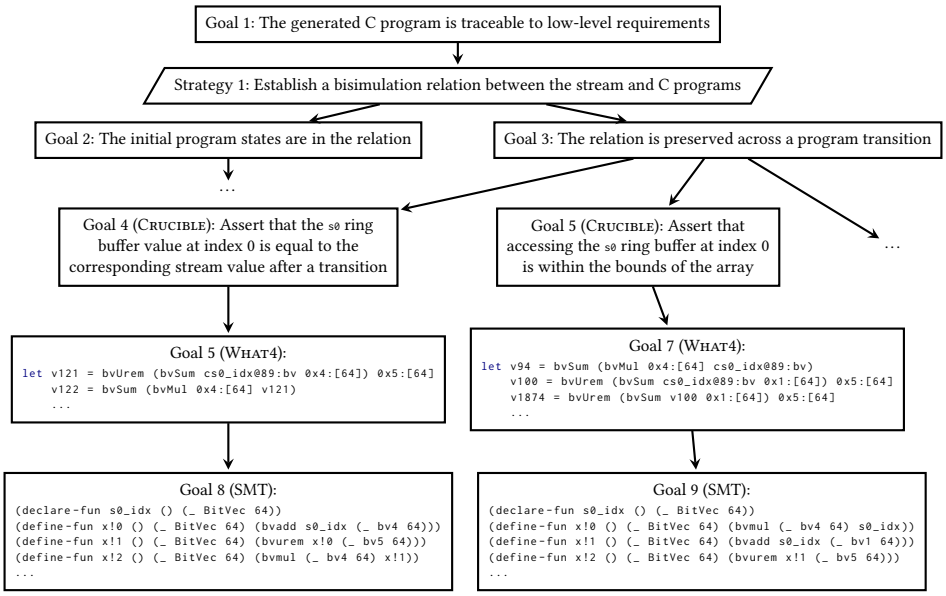


Fig. 6. A sketch of a GSN diagram presenting a CopilotVerifier assurance case for the thermostat programs from Figures 1 and 3. For brevity's sake, only a subset of goals are shown.

portion of a proof can be broken down into lower- and lower-level requirements until eventually reaching SMT, with a chain of evidence linking each intermediate step.

Using the evidence that CopilotVerifier produces, one can construct a complete assurance case that is more amenable to certification. Figure 6 shows a sketch of what an assurance case would look like for the thermostat programs from Figures 1 and 3. The evidence is captured using *Goal Structuring Notation* (GSN) [Kelly and Weaver 2004], which presents each goal in a CopilotVerifier safety case in a way that emphasizes the relationship between high-level parent goals and lower-level child goals. Aside from proof goals, a GSN diagram can also be used to document assumptions that the verifier makes during verification, such as those discussed in Section 6.

The process of making CopilotVerifier's evidence be acceptable for auditors is ongoing work. We plan to work with auditors to iteratively improve formats and explanations towards the goal of providing convincing evidence in a suitable format.

6 DESIGN TRADEOFFS

When designing CopilotVerifier, our goal was to create a verification tool that achieves high assurance at modest cost. This section explains the tradeoffs involved in achieving this.

6.1 Trusted Computing Base (TCB)

We trust the underlying C toolchain. CopilotVerifier relies on Clang to compile C into LLVM bitcode, which becomes the basis for producing the semantics of the C file. Bugs in Clang may affect the soundness of the verifier. We consider this risk mitigated by the fact that Clang is a well-tested compiler and that CopilotC99 targets a well-understood subset of C, reducing the likelihood of triggering compiler bugs. The Copilot developers have experimented using CompCert to verify the compiled binary code [Goodloe 2016; Leroy 2009], which would remove this portion of the TCB. However, CompCert does not target many of the processors that the Copilot users utilize.

We trust that Crucible's LLVM backend faithfully encodes the semantics of LLVM bitcode. Errors in this part of Crucible could affect the soundness of the verifier. To justify this trust, we note that Crux [Tomb 2020], a verification tool also based on Crucible, has been tested on a large number of C verification problems from the SV-COMP verification competition [Scott et al. 2021].

We also trust that the CopilotTheorem library accurately encodes the semantics of Copilot stream programs. At present, we do not have a robust way to test these semantics beyond careful manual engineering and comparison with the Copilot interpreter.

Because of the way we model trigger functions, we make a number of implicit assumptions about how the implementations of those functions must behave. In particular, we assume that trigger functions do not modify any memory under the control of the Copilot program, including its ring buffers and stack. We also assume that the trigger functions are memory-safe and do not perform any undefined behavior. Responsibility for enforcing these assumptions lies with the user who supplies definitions for the trigger functions.

6.2 C Compiler Optimizations

CopilotVerifier assumes that all streams in a Copilot spec have corresponding static, global arrays in the generated LLVM bitcode. This assumption simplifies the work done by CopilotVerifier, as it can associate each stream with a distinct, top-level array.

While this assumption is safe to make when the generated C code is compiled with low optimization settings, it is *not* safe at higher settings. For instance, consider a stream containing a single value, which the verifier assumes to be translated to an array of length 1. At `-O1` or higher, Clang replaces length-1 static arrays with scalar values, which breaks the verifier's assumptions.

The verifier mitigates these issues by always invoking Clang with `-O0`. This makes the generated LLVM bitcode more predictable at the expense of only verifying code at lower levels of optimization.

6.3 Limits of Automation

CopilotVerifier aims to be as push-button as possible. However, in exceptional circumstances, a statement may be true but cannot be automatically verified: that is, the verifier is *sound, but not complete*. Users may need to alter the original spec in order to make it amenable to verification.

6.3.1 Floating-Point Support. Copilot provides a variety of floating-point operations, including transcendental functions and other primitive functions. There is limited SMT solver support for floating-point values, however, and even less support for robustly handling special functions. Nevertheless, we wish to have *some* level of support, as many Copilot monitors make essential use of floating-point operations. For instance, detecting if an unmanned aircraft system is *well clear* [Upchurch et al. 2014] uses the square-root function to compute the lengths of vectors.

CopilotVerifier treats floating-point operations as uninterpreted functions, leaving the semantics of the operations abstract. This is sound, as the verifier need only demonstrate that a Copilot program applies the same floating-point operations as the corresponding C program, and in the same order. The downside to this approach is that reasoning about floating-point operations is somewhat fragile. The verifier relies on the Clang not optimizing the operations to the point where they differ from the stream program's operations. As an example, consider `ctemp` from Figure 1:

```
ctemp :: Stream Float
ctemp = (unsafeCast temp * constant (150.0/255.0)) - constant 50.0
```

A subtlety of this definition is that the division operation occurs before being lifted into a stream with the `constant` function. As a result, the reified stream in Figure 2 contains `0.5882353`, the result of the division. If we instead express this as the result of dividing two stream values:

```
ctemp = (unsafeCast temp) * (constant 150.0 / constant 255.0)) - constant 50.0
```

then the reified Copilot spec would *not* evaluate the result of dividing the two streams to `0.5882353`. On the other hand, the generated C code would contain `150.f / 255.0f`, and C compilers will perform constant folding on this, even on low optimization levels. As a result, the stream semantics would contain an uninterpreted division function but the C semantics would not, leading an SMT solver to conclude that they are not equivalent.

We take some measures to mitigate this issue, such as invoking C compilers on low optimization settings (see Section 6.2) and avoiding the use of the `--fast-math` flag. Under these settings, C compilers, and Clang in particular, are more reluctant to rearrange floating-point code, which results in more programs successfully verifying. Nevertheless, the example above shows that this is not foolproof, and users may need to rearrange code to make the operations align in just the right way.

6.3.2 Invariants for Partial Operations. As detailed in Section 4.3.4, the verifier has a mode for aborting the proof early if it finds a partial operation applied to undefined inputs. In this mode, the verifier does not try to infer invariants needed to make operations well defined. For example:

```
stream :: Stream Int32
stream = extern "stream" Nothing
```

```
spec :: Spec
spec = trigger "streamAdd"
  ((stream + 1) == 2) [arg stream]
```

When the abort-early mode for partial operations is enabled, this example will fail to verify, as `stream + 1` could result in signed integer overflow if a value in `stream` is equal to the maximum value of an `Int32`. It is possible, however, that the applications that use this monitor maintain the invariant that the values in `stream` are always less than the maximum `Int32` size. If that is the case, the user must communicate this invariant by declaring a property in the spec:

```
spec :: Spec
spec = do prop "notInt32Max" (forall (stream < constI32 maxBound))
          trigger "streamAdd" ...
```

Here, `prop` is used to declare a property named `notInt32Max`, and the `forall` combinator is used to express that the property should hold for all elements in the stream. The verifier must then be instructed to assume the `notInt32Max` property during verification so that the addition is well defined.

7 RELATED WORK

Copilot was originally based on Lustre [Caspi et al. 1987] and LOLA [D'Angelo et al. 2005]. Other similar RV frameworks include RMOR [Havelund 2008], which compiles to C, and Proteus [McClelland 2021; McClelland et al. 2021], which compiles to C++. CopilotVerifier shares similarity with other translation validation–based compilers, such as previous efforts to translate SIGNAL [Pnueli et al. 1998] and Simulink [Ryabtsev and Strichman 2009] to C, as well as the Alive2 bounded translation validation tool for LLVM [Lopes et al. 2021].

Most other RV frameworks do not verify their compiled code against the specifications. A notable exception is Lustre, which boasts a verified compiler named Vélus [Bourke et al. 2017, 2019, 2021]. Vélus is an ambitious project that verifies the Lustre compilation pipeline within the Coq theorem prover by building on CompCert [Leroy 2009]. This is in contrast to CopilotVerifier's translation validation approach, which only allows verifying individually translated programs in isolation.

Vélus first translates Lustre's stream functions into a synchronous transition code (STC) language based on state transitions and values rather than streams. STC is, in turn, translated into an object-oriented language (Obc) with each Lustre node represented as a class with its own memory, possessing a reset method performing initialization and a step method to process the next instant in time. Obc is translated into CompCert's Clight, which is then translated into assembly. Coq is used to manually prove an equivalence at each translation step. The proofs are aided by a memory

semantics given in terms of streams of memory trees. The culmination of the effort is a bisimulation theorem relating the behavior of a Lustre node and the generated assembly code.

An approach grounded in manual proof can be more complete than CopilotVerifier's automated approach. For instance, the difficulties that we encountered with floating-point operations (Section 6) could be surmounted using a Coq formalization of floating-point computation [Boldo and Melquiond 2011; Ramananandro et al. 2016]. On the other hand, adapting Vélus to build a verified Copilot compiler would be non-trivial. Although the core stream languages are very similar, Vélus introduces complexities such as the object-oriented intermediate language to accommodate Lustre features not present in Copilot. Alternatively, one could build a verified Copilot compiler from scratch, but in either case, considerable resources would need to be dedicated to the effort. Our approach to Copilot verification was influenced by the need to assure an existing compiler for a DSL that is in use at NASA, and to accomplish our task under significant resource and time constraints, hence demonstrating the applicability of the technique to many industrial projects.

Previous versions of Copilot [Pike et al. 2012, 2013] performed limited verification of C code by compiling with two different backends and verifying the equivalence of the generated programs using CBMC [Clarke et al. 2004]. This approach can potentially catch many of the same issues as CopilotVerifier, but it does not prove that the C programs match the original Copilot monitors.

Current versions of Copilot use property-based testing [Claessen and Hughes 2000; Fink and Bishop 1997] for added assurance. Although techniques such as property-based testing could catch some of the same bugs that CopilotVerifier was able to detect, it would be unlikely to detect some of the memory safety bugs uncovered by CopilotVerifier (some of which went unnoticed until recently [GitHub 2022c]).

There have been efforts in Copilot to generate C code that includes Hoare-logic–style contracts as function comments, where the contracts are derived from parts of the stream program that the C function should implement. The Frama-C static analyzer [Cuoq et al. 2012] is then used to prove that the C code satisfies the contracts [Goodloe 2016]. This approach is not fully automated: manual edits are sometimes required for the generated contracts to pass the analysis checks.

8 CONCLUSIONS

We have presented CopilotVerifier, a verifier that proves correspondences between high-level Copilot monitors and low-level C code generated by the Copilot compiler. Our aim was to increase confidence in the Copilot compiler while working within a realistic engineering budget. Through careful design choices and strategic use of existing tools, we have achieved this goal.

In the future, we plan to use CopilotVerifier to construct assurance cases that are amenable to certification. Specifically, the NASA tool Ogma uses Copilot to produce complete monitoring applications for NASA Core Flight System, Robot Operating System 2, and FPrime [Perez et al. 2022]. We plan to extend Ogma to leverage CopilotVerifier to produce evidence of assurance for the applications generated.

ACKNOWLEDGMENTS

This manuscript has been authored by Ivan Perez, an employee of KBR under Prime Contract No. 80ARC020D0010 with the NASA Ames Research Center. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views, either expressed or implied, of any of the funding organizations. The United States Government retains, and by accepting the article for publication, the publisher acknowledges that the United States Government retains, a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this work, or allow others to do so, for United States Government purposes.

REFERENCES

Sylvie Boldo and Guillaume Melquiond. 2011. Flocq: A unified library for proving floating-point algorithms in Coq. In *2011 IEEE 20th Symposium on Computer Arithmetic*. IEEE, 243–252. https://doi.org/10.1109/ARITH.2011.40

Brett Boston, Samuel Breese, Joey Dodds, Mike Dodds, Brian Huffman, Adam Petcher, and Andrei Stefanescu. 2021. Verified cryptographic code for everybody. In *CAV 2021*. https://doi.org/10.1007/978-3-030-81685-8_31

Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. 2017. A Formally Verified Compiler for Lustre. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, 586–601. https://doi.org/10.1145/3140587.3062358

Timothy Bourke, Lélio Brun, and Marc Pouzet. 2019. Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset. *Proc. ACM Program. Lang.* 4, POPL, Article 44 (dec 2019), 29 pages. https://doi.org/10.1145/3371112

Timothy Bourke, Paul Jeanmaire, Basile Pesin, and Marc Pouzet. 2021. Verified Lustre Normalization with Node Subsampling. *ACM Trans. Embed. Comput. Syst.* 20, 5s, Article 98 (sep 2021), 25 pages. https://doi.org/10.1145/3477041

P. Caspi, D. Pialiud, N. Halbwachs, and J. Plaice. 1987. LUSTRE: a Declarative Language for Programming Synchronous Systems. In *14th Symposium on Principles of Programming Languages*. 178–188. https://doi.org/10.1145/41625.41641

David Thrane Christiansen, Iavor S. Diatchki, Robert Dockins, Joe Hendrix, and Tristan Ravitch. 2019. Dependently Typed Haskell in Industry (Experience Report). *Proc. ACM Program. Lang.* 3, ICFP, Article 100 (July 2019), 16 pages. https://doi.org/10.1145/3341704

Andrey Chudnov, Nathan Collins, Byron Cook, Joey Dodds, Brian Huffman, Colm MacCárthaigh, Stephen Magill, Eric Mertens, Eric Mullen, Serdar Tasiran, Aaron Tomb, and Eddy Westbrook. 2018. Continuous Formal Verification of Amazon s2n. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 430–446. https://doi.org/10.1007/978-3-319-96142-2_26

Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. 268–279. https://doi.org/10.1145/351240.351266

Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A tool for checking ANSI-C programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 168–176. https://doi.org/10.1007/978-3-540-24730-2_15

Darren Cofer. 2021. Unintended Behavior in Learning-Enabled Systems: Detecting the Unknown Unknowns. In *2021 IEEE/AIAA 40th Digital Avionics Systems Conference (DASC)*. 1–7. https://doi.org/10.1109/DASC52595.2021.9594406

Darren Cofer, Isaac Amundson, Ramachandra Sattigeri, Arjun Passiand Christopher Boggs, Eric Smith, Limei Gilham, Taejoon Byun, and Sanjai Rayadurgam. 2020. Run-Time Assurance for Learning-Enabled Systems. In *NASA Formal Methods: 12th International Symposium, NFM 2020, Moffett Field, CA, USA, May 11–15, 2020, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 361–368. https://doi.org/10.1007/978-3-030-55754-6_21

Darren Cofer and Steven Miller. 2014. DO-333 Certification Case Studies. In *NASA Formal Methods*, Julia M. Badger and Kristin Yvonne Rozier (Eds.). Springer International Publishing, Cham, 1–15. https://doi.org/10.1007/978-3-319-06200-6_1

National Research Council. 2014. *Autonomy Research for Civil Aviation: Toward a New Era of Flight*. The National Academies Press.

Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-c. In *International conference on software engineering and formal methods*. Springer, 233–247. https://doi.org/10.1007/978-3-642-33826-7_16

B. D'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, Zohar Manna, B. Finkbeiner, H. Spima, and S. Mehrotra. 2005. LOLA: Runtime Monitoring of Synchronous Systems. In *12th International Symposium on Temporal Representation and Reasoning*. IEEE, 166–174. https://doi.org/10.1109/TIME.2005.26

Robert Dockins, Adam Foltzer, Joe Hendrix, Brian Huffman, Dylan McNamee, and Aaron Tomb. 2016. Constructing Semantic Models of Programs with the Software Analysis Workbench. In *Verified Software. Theories, Tools, and Experiments*, Sandrine Blazy and Marsha Chechik (Eds.). Springer International Publishing, Cham, 56–72. https://doi.org/10.1007/978-3-319-48869-1_5

Yliès Falcone, Klaus Havelund, and Giles Reger. 2013. A Tutorial on Runtime Verification. In *Engineering Dependable Software Systems*, Manfred Broy, Doron A. Peled, and Georg Kalus (Eds.). NATO Science for Peace and Security Series, D: Information and Communication Security, Vol. 34. IOS Press, 141–175. https://doi.org/10.3233/978-1-61499-207-3-141

George Fink and Matt Bishop. 1997. Property-Based Testing: A New Approach to Testing for Assurance. *SIGSOFT Softw. Eng. Notes* 22, 4 (jul 1997), 74–80. https://doi.org/10.1145/263244.263267

Issue on Copilot GitHub. 2021a. Basic example involving structs generates invalid C code. https://github.com/Copilot-Language/copilot/issues/275

Issue on Copilot GitHub. 2021b. C translation doesn't correctly select operations based on types. https://github.com/Copilot-Language/copilot/issues/263

Issue on Copilot GitHub. 2021c. copilot-c99: C99 and interpretation of signum behave inconsistently for 0 and NaN. https://github.com/Copilot-Language/copilot/issues/278

Issue on Copilot GitHub. 2021d. Delaying streams of arrays or structs produces C code that fails to compile with Clang. https://github.com/Copilot-Language/copilot/issues/276

Issue on Copilot GitHub. 2021e. Generated C code accesses stream buffer out of bounds. https://github.com/Copilot-Language/copilot/issues/238

Issue on Copilot GitHub. 2022a. copilot-c99: appending values to a stream of arrays produces incorrect C99 code. https://github.com/Copilot-Language/copilot/issues/314

Issue on Copilot GitHub. 2022b. copilot-c99: array has incomplete element type error when declaring stream with array of structs. https://github.com/Copilot-Language/copilot/issues/373

Issue on Copilot GitHub. 2022c. copilot-c99: Streams of arrays result in C code with undefined behavior. https://github.com/Copilot-Language/copilot/issues/386

Alwyn Goodloe. 2016. Challenges in high-assurance runtime verification. In *International Symposium on Leveraging Applications of Formal Methods*. Springer, 446–460. https://doi.org/10.1007/978-3-319-47166-2_31

Alwyn Goodloe and Lee Pike. 2010. *Monitoring Distributed Real-Time Systems: A Survey and Future Directions*. Technical Report NASA/CR-2010-216724. NASA Langley Research Center.

Klaus Havelund. 2008. Runtime verification of C programs. In *Testing of Software and Communicating Systems*. Springer, 7–22. https://doi.org/10.1007/978-3-540-68524-1_3

Joe Hendrix, Ben Selfridge, and Robert Dockins. 2020. What4: New Library to Help Developers Build Verification and Program Analysis Tools. https://galois.com/blog/2020/07/what4-new-library-to-help-devs-build-verification-program-tools/

Tim Kelly and Rob Weaver. 2004. The goal structuring notation–a safety argument notation. In *Proceedings of the dependable systems and networks 2004 workshop on assurance cases*, Vol. 6. Citeseer.

Ramana Kumar, Magnus O Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. *ACM SIGPLAN Notices* 49, 1 (2014), 179–191. https://doi.org/10.1145/2578855.2535841

Jonathan Laurent, Alwyn Goodloe, and Lee Pike. 2015. Assuring the guardians. In *Runtime Verification*. Springer, 87–101. https://doi.org/10.1007/978-3-319-23820-3_6

Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

Nuno P Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: bounded translation validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 65–79. https://doi.org/10.1145/3453483.3454030

Brian McClelland. 2021. *Adding Runtime Verification to the Proteus Language*. Ph. D. Dissertation. California State University, Northridge.

Brian McClelland, Daniel Tellier, Meyer Millman, Kate Beatrix Go, Alice Balayan, Michael J Munje, Kyle Dewey, Nhut Ho, Klaus Havelund, and Michel Ingham. 2021. Towards a systems programming language designed for hierarchical state machines. In *2021 IEEE 8th International Conference on Space Mission Challenges for Information Technology (SMC-IT)*. IEEE, 23–30. https://doi.org/10.1109/SMC-IT51442.2021.00010

AFE87 Project Members. 2020. *AFE 87 - Machine Learning*. Technical Report. Aerospace Vehicle Systems Institute.

Ivan Perez, Frank Dedden, and Alwyn Goodloe. 2020. *Copilot 3*. Technical Report. NASA Langley Research Center.

Ivan Perez, Anastasia Mavridou, Tom Pressburger, Alwyn Goodloe, and Dimitra Giannakopoulou. 2022. Automated Translation of Natural Language Requirements to Runtime Monitors. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 387–395.

Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. 2010. Copilot: a hard real-time runtime monitor. In *International Conference on Runtime Verification*. Springer, 345–359. https://doi.org/10.1007/978-3-642-16612-9_26

Lee Pike, Nis Wegmann, Sebastian Niller, and Alwyn Goodloe. 2012. Experience report: a do-it-yourself high-assurance compiler. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*. 335–340. https://doi.org/10.1145/2398856.2364553

Lee Pike, Nis Wegmann, Sebastian Niller, and Alwyn Goodloe. 2013. Copilot: monitoring embedded systems. *Innovations in Systems and Software Engineering* 9, 4 (2013), 235–255. https://doi.org/10.1007/s11334-013-0223-x

Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation validation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 151–166. https://doi.org/10.1007/BFb0054170

Tahina Ramananandro, Paul Mountcastle, Benoît Meister, and Richard Lethin. 2016. A unified Coq framework for verifying C programs with floating-point computations. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*. 15–26. https://doi.org/10.1145/2854065.2854066

RTCA 2011. Formal Methods Supplement to DO-178C and DO-278A. RTCA, Inc.. RCTA/DO333.

RTCA. 2011. Software Considerations in Airborne Systems and Equipment Certification. RTCA, Inc.. RCTA/DO-178C.

Michael Ryabtsev and Ofer Strichman. 2009. Translation validation: From simulink to c. In *International Conference on Computer Aided Verification*. Springer, 696–701. https://doi.org/10.1007/978-3-642-02658-4_57

SAE International. 2010. Guidelines For Development Of Civil Aircraft and Systems. SAE International. ARP4754A.

Ryan Scott, Robert Dockins, Tristan Ravitch, and Aaron Tomb. 2021. Crux: Symbolic Execution Meets SMT-based Verification (Competition Contribution). https://doi.org/10.5281/zenodo.6147218

Ryan G. Scott, Mike Dodds, Ivan Perez, Alwyn E. Goodloe, and Robert Dockins. 2023. *Artifact for "Trustworthy Runtime Verification via Bisimulation (Experience Report)"*. https://doi.org/10.5281/zenodo.7978326

Aaron Tomb. 2020. Crux: Introducing Our New Open-Source Tool for Software Verification. https://galois.com/blog/2020/10/crux-introducing-our-new-open-source-tool-for-software-verification/

Jason M Upchurch, Cesar A Munoz, Anthony J Narkawicz, James P Chamberlain, and Maria C Consiglio. 2014. *Analysis of Well-Clear Boundary Models for the Integration of UAS in the NAS*. Technical Report. NASA Langley Research Center.

Lucas Wagner, Alain Mebsout, Cesare Tinelli, Darren Cofer, and Konrad Slind. 2017. Qualification of a Model Checker for Avionics Software Verification. In *NASA Formal Methods*, Clark Barrett, Misty Davies, and Temesghen Kahsai (Eds.). Springer International Publishing, Cham, 404–419. https://doi.org/10.1007/978-3-319-57288-8_29

Received 2023-03-01; accepted 2023-06-27