Experience Report: a Do-It-Yourself High-Assurance Compiler

Lee Pike
Galois, Inc.
leepike@galois.com

Nis Wegmann
University of Copenhagen
niswegmann@gmail.com

Sebastian Niller
Unaffiliated
sebastian.niller@gmail.com

Alwyn Goodloe
NASA
a.goodloe@nasa.gov

Abstract

Embedded domain-specific languages (EDSLs) are an approach for quickly building new languages while maintaining the advantages of a rich metalanguage. We argue in this experience report that the "EDSL approach" can surprisingly ease the task of building a high-assurance compiler. We do not strive to build a fully formally-verified tool-chain, but take a "do-it-yourself" approach to increase our confidence in compiler-correctness without too much effort. *Copilot* is an EDSL developed by Galois, Inc. and the National Institute of Aerospace under contract to NASA for the purpose of runtime monitoring of flight-critical avionics. We report our experience in using type-checking, QuickCheck, and model-checking "off-the-shelf" to quickly increase confidence in our EDSL tool-chain.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

The "do-it-yourself" (DIY) culture promotes individuals to design and craft objects on their own, without relying on outside experts. DIY construction should be inexpensive with easy-to-access materials. Ranging from hobbyist electronics[1] to urban farming to fashion, DIY is making somewhat of a resurgence across the United States.

We see no reason why DIY culture should not also extend to compilers, and in particular, to high-assurance compilers. By *high-assurance*, we mean a compiler that comes with compelling evidence that the source code and object code have the same operational semantics.

High-assurance compilers development has traditionally required years of effort by experts. A notable early effort was the CLI Stack, of which a simple verified compiler was one part [18]. The CLI Stack was verified by the precursor to the ACL2 theorem prover. The most recent instance is CompCert, which compiles a subset of C suitable for embedded development to machine code for a number of targets [17]. CompCert is formally verified in the Coq theorem-prover—indeed, CompCert is written in Coq's specification language. While CompCert achieves the highest levels of assurance, generating the evidence comes at a steep price, since it relies on manually interacting with a theorem-prover. Neither the CLI Stack nor CompCert are DIY projects: building them requires relatively esoteric skills that combine interactive theorem-proving and multiple engineer-years of verification effort.

In this experience report, we argue that by leveraging functional languages and off-the-shelf verification tools, we can accumulate significant evidence of correctness at a fraction of the cost and without the specialized know-how required by interactive verification approaches.[2]

The case-study of our approach is the Copilot language and toolset, developed by Galois, Inc. and the National Institute of Aerospace under contract to NASA. Copilot is a stream language for generating embedded C-code software monitors for system properties. Copilot itself is not comparable to a verified compiler like CompCert: Copilot back-ends stop at the C level, where CompCert starts. Verifying C semantics against the semantics of a machine model is extraordinarily difficult. Still, for high-level languages, we can do much better than the status quo.

Specifically, we employ three not-so-secret weapons from the functional languages and formal methods communities in our work.

1. *Embedded DSLs*: We implement Copilot as an embedded domain-specific (EDSL) language [16] within Haskell.
2. *Sub-Turing complete languages*: Copilot is targeted at embedded programming, therefore we focus on the class of programs that are computable in constant time and constant space.
3. *A verifying compiler*: CompCert typifies a *verified compiler* approach in which the compiler itself is proved correct. A *verifying compiler* is one that provides evidence that a specific compilation is correct [20]. We borrow from this second approach. (We emphasize that this report is about assurance of the EDSL compiler itself, not about the functional correctness of programs written in the EDSL.)

While the approaches we describe are known within the functional programming and formal methods communities, the purpose of this experience report is to demonstrate the engineering ease in putting them to use. In particular, the EDSL approach is well-known for quickly prototyping new languages, but the reader should have some level of skepticism that they are appropriate for high-assurance development; we hope to dispel that skepticism. Furthermore, there is nothing special about the Copilot language with respect to assurance. We hope to convince the reader that the approach we have taken can be applied broadly to new language design.

[1] This even includes full-featured unpiloted air vehicles! See `http://diydrones.com/`.

[2] Our work perhaps is an instance of the *Pareto Principle* with respect to compiler assurance. The principle states that 80% of effects arise from 20% of the causes in a wide variety of empirical observations [22].

Outline. In Section 2, we briefly introduce Copilot (we assume some familiarity with Haskell syntax). The heart of the report is Section 3 in which we describe our "lessons learned" for easily generating evidence of correctness. We briefly mention related work in Section 4, and make concluding remarks in Section 5.

2. The Copilot Language & Toolset

From 2009-2011, NASA contracted Galois, Inc. to research the possibility of augmenting complex aerospace software systems with *runtime verification* (RV). RV is a family of approaches that employ *monitors* to observe the behavior of an executing system and to detect if it is consistent with a formal specification. A monitor implementation should be simple and direct and serve as the last line of defense for the correctness for the system. The need for aerospace RV is motivated by recent failures in commercial avionics and the space shuttle [12].

Our answer to the contract goals was Copilot, an EDSL to generate embedded monitors.[3] The Copilot language itself, focusing on its RV uses for NASA, has been previously described [21]. We will very briefly introduce Copilot in this paper; our focus is more specifically about compiler correctness.

One research challenge of the project was phrased as, "Who watches the watchmen?" meaning that if the RV monitor is the last line of defense, then it must not itself fail or worse, introduce unintended faults itself! Nonetheless, because the primary goal of the project was to implement an RV system and to field-test it, few resources were available for assuring the correctness of the Copilot compiler itself. Our approach was born out of necessity.

Copilot's expression language. In the following, we briefly and informally introduce Copilot's expression language. One design goal for Copilot is to use a familiar syntax and model of computation; doing so is a first step in reducing specification errors. The Copilot language mimics both the syntax and semantics of lazy lists (which we call *streams*) in Haskell. One notable exception though is that operators are automatically promoted point-wise to the list level, much like in Lustre, a declarative language for embedded programming [13]. For example, the Fibonacci sequence modulo 2^{32} can be written as follows in Haskell:

```
fib :: [Word32]
fib = [0,1] ++ zipWith (+) fib (drop 1 fib)
```

In Copilot, the equivalent definition is the following:

```
fib :: Stream Word32
fib = [0,1] ++ (fib + drop 1 fib)
```

Copilot overloads or redefines many standard operators from Haskell's Prelude Library. Here is a Haskell and equivalent Copilot function that implements a latch (flip-flop) over streams—the output is the XOR of the input stream and the latch's previous output. For example, for the input stream

```
T, F, F, T, F, F, T, F, F, ...
```

`latch` generates the stream

```
F, T, T, T, F, F, F, T, T, T, ...
```

In Haskell, `latch` can be defined

```
latch :: [Bool] -> [Bool]
latch x = out x
  where
  out ls   = [False] ++ zipWith xor ls (out ls)
  xor n m  = (n || m) && not (n && m)
```

[3] Copilot source code is available at `http://leepike.github.com/Copilot/` and is licensed under the BSD3 license.

and then in Copilot (`xor` is a built-in operator for Copilot):

```
latch :: Stream Bool -> Stream Bool
latch x = out
  where out = [False] ++ x `xor` out
```

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 *sampling* values. Concretely, the values are from in-scope symbols in a linked C program. Copilot samples variables, arrays, and side-effect free functions (sampling arbitrary structures is future work). The operator `extern` takes a C symbol and a possible list of values for use by the interpreter. For example, the following stream samples the C variable `e0` of type `uint8_t` to create each new stream index. For example, if `e0` takes the values `2, 4, 6, ...` the stream `ext` has the values `1, 3, 7, 13, ...`.

```
ext :: Stream Word8
ext = [1] ++ (ext + extern "e0" interp)
  where interp = Just [2,4..]
```

We make the design decision to build interpreter values for external values into the language. (If the user wishes not to provide interpreter values, the constructor `Nothing` can be used.)

Sampling arrays and functions are similar. For example, the following stream samples an array with the prototype `uint32_t arr[3]`:

```
arr :: Stream Word32
arr = externArray "arr" idx 3 interp
  where idx :: Stream Word8
        idx    = [0] ++ (idx + 1) `mod` 2
        interp = Just (repeat [0,1,2])
```

The interpreter takes a list of lists to represent possible array values.

Effects. Copilot has exactly one mechanism for output called *triggers*. For example, consider the following example trigger:

```
trigger "trig" (fib `mod` 2 == 0)
  [ arg fib, arg (latch fib) ]
```

A trigger has a guard that is a Boolean-valued Copilot stream, and a list of arguments, which are Copilot expressions. A trigger is *fired* exactly when its guard (stating that the current value from the `fib` stream is even in this case) is true. A trigger's implementation is a C function with a `void` return type that takes the current values of the trigger arguments as arguments. For example, given the definition of `fib` and `latch` above, the prototype of the C-function implementing the trigger is

```
void trig(uint32_t, bool);
```

The definition of a trigger is implementation-dependent and up to the programmer to implement.

Copilot's toolchain. Copilot's toolchain is depicted in Figure 1. The Copilot language package reifies a Copilot program (i.e., turns a recursive structure into explicit graphs via observable sharing [11]—Copilot is deeply embedded in Haskell) and does domain-specific type-checking (∼1200 lines of code or LOCs). Copilot programs are then translated into a core language (∼900 LOCs). The core package contains an interpreter (∼300 LOCs) as well as a custom QuickCheck engine and test harness for testing interpreter output against one of the back-ends (∼400 LOCs). Copilot's language package is explicitly *Trustworthy Haskell*, as there is a single instance of `unsafeCoerce` to implement observable sharing. Copilot's core language is written in *Safe Haskell* [10].

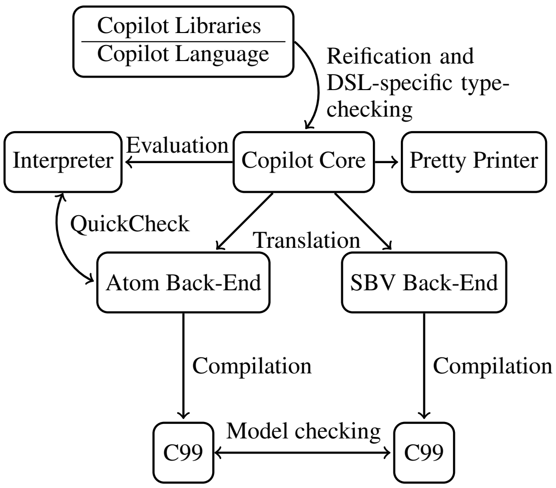

Figure 1. The Copilot toolchain.

The back-ends (each of which is ∼1100 LOCs) translate a Copilot core program into the language of another Haskell DSL for code generation. We use the Atom (∼2500LOCs)[4] [14] and SBV (∼4500LOCs)[5] packages for code generation, both of which generate a strict subset of C99 embedded code that is constant-memory and nearly constant-time.

Atom is an EDSL originally designed by Tom Hawkins at Eaton Corp. for synthesizing real-time embedded control systems from high-level specifications. The language provides scheduling constructs, obviating the need for a real-time operating system when cooperative scheduling is sufficient.

Symbolic Bit Vectors (SBV) is an EDSL developed by Levent Erkök. The primary focus of SBV is to express and reason about bit-level Haskell programs. In particular, the language provides tight integration with satisfiability modulo theories (SMT) solvers (e.g., Yices [8]) for automatic proofs and to check for satisfiability. The EDSL also contains a C-code generator which we use. Other features of the language include test-case generation and automated synthesis.

A separate package (∼100 LOCs) generates a driver for the CBMC model-checker [7], which we use to check the equivalence between the C code generated by each back-end.

3. Lessons Learned: Quick and Easy Correctness Evidence

In the following, we describe some "lessons-learned" in quickly and easily building assurance into an EDSL compiler.

Lesson: Turing-complete macros, small, Turing-incomplete languages. C-like languages treat macros as a second-class feature—they are just textual substitution. Lisp-like languages take the converse approach, treating macros as a first-class datatype, so macros are on par with (Turing-complete) programming. These are two extremes, but they largely represent the status of macro programming.

EDSLs, however, treat meta-programming as first-class, and programming as second-class! The difference in emphasis of EDSLs is because the embedded language is a datatype within its host language (we assume a deep-embedding of the DSL [11]). The difference affects how one programs using an EDSL. Practically, one spends very little time directly using the operators of the EDSL itself but rather, one generates EDSL programs using combinators from the host language.

Embedded system programming, with time and memory constraints, does not require the full power of a general-purpose Turing-complete language [5]. But a Turing-complete *macro language* affords benefits in code-reuse and library development. With an EDSL, one can have his cake and eat it too: Arbitrarily complex combinators over the EDSL can be written, but then a simple core language can be reasoned about.

Reasoning about the correctness of sub-Turing-complete languages is easier than general-purpose languages. For example, a verifying compiler for a cryptographic DSL leveraged the ability to automatically generate measures to formally prove termination of programs written in the language [20]. Conversely, Sassaman *et al.* argue that a principal origin of insecurity in computer systems is due to Turing-complete (or more generally, too powerful) data-description languages [24].

```
data Expr a where
  -- Constants
  Const :: Type a -> a -> Expr a
  -- Stream constructors
  Drop  :: Type a -> Int -> Id -> Expr a
  -- Let expressions
  Local :: Type a -> Type b -> Name -> Expr a
        -> Expr b -> Expr b
  Var   :: Type a -> Name -> Expr a
  -- Operators
  Op1   :: Op1 a b -> Expr a -> Expr b
  Op2   :: Op2 a b c -> Expr a -> Expr b -> Expr c
  Op3   :: Op3 a b c d -> Expr a -> Expr b
        -> Expr c -> Expr d
  -- Externals
  ExternVar
    :: Type a -> Name -> Maybe [a] -> Expr a
  ExternFun
    :: Type a -> Name -> [UExpr]
    -> Maybe (Expr a) -> Maybe Int -> Expr a
  ExternArray
    :: Integral a => Type a -> Type b
    -> Name -> Int -> Expr a -> Maybe [[b]]
    -> Maybe Int -> Expr b

-- Untyped streams
data UExpr = forall a. UExpr
  { uExprType :: Type a
  , uExprExpr  :: Expr a }
```

Figure 2. The core Copilot expression language abstract syntax.

The core language of Copilot is both small and unpowerful: as noted, only programs requiring a constant amount of space can be written in Copilot. In Figure 2 is the generalized abstract datatype (GADT) [25] that is the abstract syntax for Copilot expressions in the core language. There are constants, the "drops" stream constructor (dropping a finite number of prefix list elements), let-expressions within Copilot for user-defined expression sharing, external program inputs, and unary, binary, and ternary operators. One final data type, `UExpr` contains existentially-typed streams that are used in argument lists. Everything else is syntactic sugar or specific operators. (The operational semantics of Copilot, given by an interpreter function over the `Expr` datatype, is about 200 LOCs.)

Despite the small size of the core language and the lack of computational power, with Haskell's parametric polymorphism and standard library combinators, we can enjoy the benefits of code reuse and abstraction in building libraries while maintaining a terse core language. For example, in our fault-tolerant voting library, the

[4] `http://hackage.haskell.org/package/atom`, BSD3 license.

[5] `http://hackage.haskell.org/package/sbv`, BSD3 license.

Boyer-Moore linear-time Majority Vote algorithm [4] is written as a Haskell function that gets expanded at compile-time into a Copilot program. Libraries for bounded linear-temporal logic, regular expressions, bounded folds, bounded scans, etc. are similarly just Copilot macros.

The idea that the macro language can be arbitrarily complex is obvious to the functional languages community, but it is a disruptive one to the embedded languages community, particularly for safety-critical systems. Typical declarative languages for embedded systems design, like Lustre [5], are not polymorphic (polymorphism is limited to a small set of pre-defined operators, like `if-then-else`).

Lesson: decomposed type-checking. Type-checking is the first defense against incorrect programs. We used a two-layer approach: let the host language enforce types where possible, and write a custom checker for type-checking that falls outside of the host language's type system. In this way, we rely on Haskell to do most of the heavy lifting.

We use GADTs to represent both the front-end abstract syntax and the core language. The use of parameterized datatypes makes the probability of unanticipated type-casts low. There are only two cases during which we escape Haskell's type system, which may lead to incorrect type-casts.

The first case is when a back-end pretty-prints C code. The correctness of such code can be determined by inspecting a small number of functions and class instances.

The second case arises during the translation from the core abstract syntax into the back-ends. Both the core language and the back-ends make use of polymorphic functions and class constraints. As a matter of software engineering, we do not want Copilot's core functions to be dependent on the classes introduced in the back-ends—doing so would require modifying the core each time a new back-end is added.

Therefore, we use the ideas of type-safe dynamic typing to translate from the core language to the back-end languages without relying on compiler extensions or unsafe functions [2]. The basic idea is to create witness functions that we pattern-match against. For example, for the class `SymWord` ("Symbolic Word") in the SBV back-end, we create the following instance datatype and an instance function mapping Copilot types to `SymWord`s:

```
data SymWordInst a = SymWord a => SymWordInst

symWordInst :: Type a -> SymWordInst a
symWordInst t =
  case t of
    Bool   -> SymWordInst
    Int8   -> SymWordInst
    ...
```

where `Type` is a phantom type containing concrete representations of Copilot's core types.

```
data Type :: * -> * where
  Bool    :: Type Bool
  Int8    :: Type Int8
  ...
```

Then during the translation, we pattern-match. For example, in translating the addition operator, we have the case:

```
transBinaryOps op = case op of
  Add t -> case W.symWordInst t of
             W.SymWordInst ->  (+)
  ...
```

The upshot is that we have created potentially partial translation functions, but type-incorrect translation is not possible.

In addition to type-checking provided by Haskell, we perform a small amount of custom type-checking (∼250 LOCs). The two classes of custom type-checking are (1) causality analysis and (2) type-checking external variables (arrays and functions). Causality analysis ensures that stream dependencies are strict. Strict dependencies are necessary when we are sampling variable values in real-time from the external world. For example, the following Copilot stream equations fail type-checking:

```
x :: Stream Word8
x = extern "ext" Nothing
y = drop 2 x
```

We also check at compile-time that streams are productive; for example, the stream definition `x = x` fails type-checking.

In addition, external variables are just strings with associated types. Therefore, we must check that the same string is not given two different types or declared to be of two different kinds of symbols (e.g,. a global variable vs. a function symbol). For example, the following two expressions, if they appear in the same Copilot program, fail type-checking:

```
x :: Stream Word8
x = extern "ext" Nothing

y :: Stream Word16
y = extern "ext" Nothing
```

Lesson: cheap front-end/back-end testing. QuickCheck [6] testing is so easy to implement and so effective that no EDSL compiler should be without it. QuickCheck can of course be used for unit testing during compiler development, but we use it to generate regression tests for the semantics of the EDSL by comparing the output of the interpreter against the Atom back-end (we plan to implement QuickCheck testing against the SBV back-end in the future).

We generate a stand-alone executable that for a user-specified number of iterations,

1. generates a random Copilot program,
2. compiles the Copilot program to C,
3. generates a `driver.c` file containing a `main` function as well as values for external variables,
4. compiles and links an executable (using *gcc*),
5. executes the program,
6. and compares its output to the output from the Copilot interpreter.

Weights can be set to determine the frequency of generating the various Copilot language constructs and streams of different types.

There are at least two approaches to generating type-correct programs. First, we can generate random programs, then filter ill-typed programs using the type-checker. Second, we can generate type-correct programs directly. We take the second approach. Generating type-correct programs is not difficult in our case: as described already, because Copilot's abstract syntax is parameterized by Haskell type variables, type-correct expression generation is straight-forward. We need only to ensure the small number of domain-specific type rules are also satisfied.

The benefit of generating type-correct programs directly is that if the generator is implemented correctly, every generated program is type-correct and will be tested. The danger, however, is that the generator may be too strict, omitting some type-correct programs from being generated and tested.

With the standard options, we generate, compile, test and pretty-print to standard output about 1,000 programs per minute. It is easy to let the QuickCheck test generator run continuously on a server, generating some million and a half vectors per day (in practice, bugs, if present, tend to appear after just 10s or 100s of generated vectors). The kinds of bugs we have caught include forgotten witness for the Atom back-end and the incorrect ordering of the state-machine with respect to the interpreter, such as outputting stream values *before* reading in external variable values. A "non-bug" we "discovered" was disagreement on floating-point values between GHC's runtime system (executing the interpreter) and *libc*. We solved this problem by just checking that floating point values are within some small constant range, noting that pathological cases may cause differences outside of a constant range without violating the IEEE floating-point standard.

Lesson: cheap back-end proofs. The verified compiler approach assumes that the compiler itself is within the *trusted computing base* (TCB)—the software that must be trusted to be correct. Consequently, it requires a monolithic approach to verification in which the compiler is verified. But what if the compiler can be removed from the TCB? Doing so can reduce the difficulty of providing assurance evidence.

This is our motivation for a proof approach of the back-ends. Recalling Figure 1, Copilot has two back-ends that generate C. We leverage a model-checker to prove the equivalence of the code generated by each back-end. Open-source model-checkers like CBMC [7] and Blast [3] use C as their specification language. In our work, we use CBMC. CBMC can prove memory-safety properties, such as division by zero, not-a-number floating-point errors, and array out of bound indexes. It can also prove arbitrary propositional formulas given in the body of `assert()` functions.

To prove equivalence between the two back-end outputs, we automatically generate a driver program that executes both back-ends for one step, compares their outputs, takes another step, compares their outputs, and so on for a user-specified number of iterations. The generated driver is of the form

```
for (i = 0; i < RNDS; i++) {
  sampleExterns();
  atom_step();
  sbv_step();
  assert(   atomStr_0 == sbvStr_0
         && atomStr_1 == sbvStr_1
         && ... );
}
```

For sampled variables (arrays, functions), we use CBMC's built-in model of nondeterminism to model arbitrary inputs to Copilot programs. CBMC proves the two programs are memory-safe and have equivalent semantics for a finite number of user-specified iterations (`RNDS`).

Model-checking works "out of the box" in our case because both back-ends generate simple code (e.g., no non-linear pointer arithmetic, no function pointers, no loops) in the state-update functions. This use of formal methods emphasizes the lesson about simple, Turing-incomplete languages from Section 3.

A proof of correspondence on the C code reduces the trust required in the Atom and SBV back-ends. Assuming the model-checker is sound, incorrectly-generated code will be claimed to be equivalent only if bugs with the same effects appear in both back-ends. In addition, memory-safety errors, even if they appear simultaneously in both programs, will be caught.

That said, one must still trust the C compiler—CompCert [17] would be a good point in this case. Furthermore, Copilot programs are expected to be executed forever (i.e., they are programs over infinite streams), which mimics the behavior of embedded software. CBMC symbolically unrolls programs either completely if possible or to a user-specified depth; it does not perform an inductive proof, so currently, we only show equivalence up to a user-set bound (`RNDS` in the code-snippet above). Using a model-checker with induction (e.g., k-induction via SAT [15, 26]) would strengthen the assurance case. Finally, note that this use of model-checking takes the verifying rather than verified-compiler approach: model-checking is done for *each* program compiled.

The kinds of bugs we have caught mostly include incorrect ordering of state-machine functions in the generated C. Because we do not yet have a QuickCheck testing infrastructure between the interpreter and the SBV back-end, we get a transitive argument that the SBV back-end is equivalent to the Atom back-end, which has evidence of matching the interpreter through the use of QuickCheck. From an evidence perspective, model-checking the back-ends reduces the required trust in the Haskell compiler/interpreter, since we check the generated artifacts. Ideally, we would have the power of an EDSL without having to trust the runtime system of the host language.

Lesson: a unified host language. Our last lesson is one obvious to the functional programming community, but novel in safety-critical languages. EDSLs are intrinsically immune to whole classes of potential compiler bugs. For example, because a separate parser, lexer, tokenizer, etc. are not necessary, EDSLs do not suffer from these front-end bugs. This assumes that the host language's front-end does not contain bugs, which for a stable well-used host language, is more likely than for a new DSL front-end.

We enjoyed two other advantages. First, translating between EDSLs in the same host language was type-safe and relatively easy since the two back-ends we use were existing EDSLs. Translating from Copilot into a back-end is a matter of converting from one abstract syntax datatype into another, never leaving the host language.

Second, the host language serves as more than a macro language: it serves as a partial build system. For example, consider the case of generating distributed Copilot programs to be run on networked processors, where we want to parameterize inputs based on the processor identifier. With an EDSL, this is no more difficult than parameterizing the `compile` function. In our experiments with NASA, we did just this to build fault-tolerant monitors [21].

4. Related Work

Our experience report builds on research in disparate fields including functional programming and EDSL design, compiler verification, and embedded safety-critical languages. In this section, we provide just a few pointers into the literature that inspired us.

Some might believe that compilers are generally bug-free (even if specific programs are buggy). Work at the University of Utah has dispelled this myth [27], having uncovered hundreds of bugs in C compilers like *gcc*, *clang*, and even the (unverified) front-end of the CompCert compiler [17]. Compiler verification is still important. Our work does not address C compilation directly, but it does reduce the risk of encountering bugs in C compilers by constraining the language to a small subset of well-defined C.

FeldSpar is an EDSL in Haskell designed for digital signal processing designed by Ericsson and Chalmers University [1]. FeldSpar's architecture and implementation is similar to Copilot's and could likely integrate the kind of assurance evidence for Copilot easily.

Researchers at the University of Minnesota have also built a family of DSLs tailored for safety-critical embedded system modeling [9]. The host language was designed so that new DSLs can be specified using attribute grammars. It appears the purpose of the language is primarily for modeling, so the work does not address compilation, and consequently does not address compiler correctness issues.

An alternative to the EDSL approach is to take a functional language and augment it with sufficient evidence to be used in safety-critical contexts directly. A consortium did just that with OCaml, rewriting the SCADE code generator [19]. SCADE is a modeling and development environment for critical embedded systems, such as avionics. The code generator was rewritten in OCaml, and it is qualifiable under DO-178B, level A, the most stringent level of tool qualification for avionics [23]. Qualifying the software required substantially reducing OCaml's runtime system and garbage collector, extensive testing, and providing "traceability" of requirements. From a formal verification perspective, the requirements are lightweight (the main direct evidence for correctness is testing), but in practice DO-178B qualification is rigorous.

5. Conclusions

Despite our experience, EDSLs are not a panacea. Copilot suffers the same problems that many EDSL implementations do. Error messages from the Haskell compiler are not domain-specific. There is no graphical development environment (common in embedded systems development). Large Haskell expressions are easy to generate, which can be expensive to interpret or compile. Copilot does not currently have a highly-optimizing back-end.

Regarding our approach to compiler assurance, there are some weaknesses. First, since the interpreter and back-ends are built on the core language, bugs in translation from the front-end will affect all the targets. While QuickCheck tests the executables against the interpreter, the model-checker only proves properties about (its interpretation of) the C source semantics. CompCert would obviously be a good choice to compile C, then. Finally, as noted in the introduction, we have focused here on evidence of correct compilation, but our implementation does not necessarily help ensure a specific program meets its specification.

These shortcomings point to future research efforts.

In summary we hoped to make two points in this report: first, that EDSLs are a viable approach for building high-assurance compilers, and second, that strong evidence can be generated with little work or expertise. With the EDSL, you do not have to write your own front-end, most type-checking is done for you, and today's off-the-shelf model-checkers are capable of checking real programs.

But don't take our word for it; do-it-yourself.

Acknowledgments

This work was supported by NASA Contract NNL08AD13T. We wish to especially thank the following individuals for advice on our work: Ben Di Vito, Paul Miner, Eric Cooper, Joe Hurd, and Aaron Tomb. Robin Morisset worked on an earlier version of Copilot. Nis Wegmann and Sebastian Niller completed this work while they were visiting researchers at the National Institute of Aerospace.

References

[1] E. Axelsson, K. Claessen, M. Sheeran, J. Svenningsson, D. Engdal, and A. Persson. The design and implementation of Feldspar - an embedded language for digital signal processing. In *Implementation and Application of Functional Languages*, volume 6647 of *LNCS*, pages 121–136. Springer, 2011.

[2] A. I. Baars and S. D. Swierstra. Typing dynamic typing. In *Intl. Conference on Functional Programming (ICFP)*, pages 157–166. ACM, September 2002.

[3] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker BLAST: Applications to software engineering. *International Journal on Software Tools for Technology Transfer*, 2007.

[4] R. S. Boyer and J. S. Moore. Mjrty: A fast majority vote algorithm. In *Automated Reasoning: Essays in Honor of Woody Bledsoe*, pages 105–118, 1991.

[5] P. Caspi, D. Pialiud, N. Halbwachs, and J. Plaice. LUSTRE: a declratative language for programming synchronous systems. In *14th Symposium on Principles of Programming Languages*, pages 178–188, 1987.

[6] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of haskell programs. In *ACM SIGPLAN Notices*, pages 268–279. ACM, 2000.

[7] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS, pages 168–176. Springer, 2004.

[8] B. Dutertre and L. D. Moura. The Yices SMT solver. Technical report, SRI, 2006.

[9] J. Gao, M. Heimdahl, and E. Van Wyk. Flexible and extensible notations for modeling languages. In *Fundamental Approaches to Software Engineering (FASE)*, volume 4422 of *LNCS*, pages 102–116. Springer Verlag, March 2007.

[10] GHC Team. The glorious Glasgow Haskell compilation system user's guide, version 7.4.1, March 2012. Available at `http://www.haskell.org/ghc/docs/latest/html/users_guide/`.

[11] A. Gill. Type-safe observable sharing in Haskell. In *Proceedings of the 2009 ACM SIGPLAN Haskell Symposium*, September 2009.

[12] A. Goodloe and L. Pike. Monitoring distributed real-time systems: A survey and future directions. Technical Report NASA/CR-2010-216724, NASA Langley Research Center, July 2010.

[13] N. Halbwachs and P. Raymond. Validation of synchronous reactive systems: from formal verification to automatic testing. In *ASIAN'99, Asian Computing Science Conference*. LNCS 1742, Springer, December 1999.

[14] T. Hawkins. Controlling hybrid vehicles with Haskell. Presentation. *Commercial Users of Functional Programming* (CUFP), 2008. Available at `http://cufp.galois.com/2008/schedule.html`.

[15] T. Kahsai, Y. Ge, and C. Tinelli. Instantiation-based invariant discovery. In *3rd NASA Formal Methods Symposium*, volume 6617 of *LNCS*, pages 192–207. Springer, 2011.

[16] D. Leijen and E. Meijer. Domain specific embedded compilers. In *Domain-Specific Languages Conference*. USENIX, 1999.

[17] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52:107–115, July 2009.

[18] J. S. Moore, editor. *Special Issue on System Verification: Journal of Automated Reasoning*, volume 5, 1989.

[19] B. Pagano, O. Andrieu, T. Moniot, B. Canou, E. Chailloux, P. Wang, P. Manoury, and J.-L. Colao. Experience report: using Objective Caml to develop safety-critical embedded tools in a certification framework. In G. Hutton and A. P. Tolmach, editors, *International Conference on Functional Programming (ICFP)*, pages 215–220. ACM, 2009.

[20] L. Pike, M. Shields, and J. Matthews. A verifying core for a cryptographic language compiler. In *Proceedings of the 6th Intl. Workshop on the ACL2 Theorem Prover and its Applications*, pages 1–10. ACM, 2006.

[21] L. Pike, S. Niller, and N. Wegmann. Runtime verification for ultra-critical systems. In *Proceedings of the 2nd Intl. Conference on Runtime Verification*, LNCS. Springer, September 2011.

[22] W. J. Reed. The Pareto, Zipf and other power laws. *Economics Letters*, 74(1):15–19, 2001.

[23] RTCA. Software considerations in airborne systems and equipment certification. RTCA, Inc., 1992. RCTA/DO-178B.

[24] L. Sassaman, M. L. Patterson, S. Bratus, and A. Shubina. The halting problems of network stack insecurity. *;login: The USENIX Magazine*, 36(6), December 2011.

[25] T. Schrijvers, S. Peyton Jones, M. Sulzmann, and D. Vytiniotis. Complete and decidable type inference for GADTs. In *International Conference on Functional Programming (ICFP)*, ICFP '09, pages 341–352. ACM, 2009.

[26] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a sat-solver. In *Formal Methods in Computer-Aided Design (FMCAD)*, volume 1954 of *LNCS*, pages 108–125. Springer, 2000.

[27] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Programming Language Design and Implementation (PLDI)*, pages 283–294. ACM, 2011.