On Verified Scala for STIX File System Embedded Code using Stainless*

Jad Hamza[1], Simon Felix[2][0000−0002−3979−128X],
Viktor Kunčak[1][0000−0001−7044−9522], Ivo Nussbaumer[2], and Filip Schramka[2]

[1] EPFL IC LARA, Lausanne, Switzerland
jad.hamza@epfl.ch viktor.kuncak@epfl.ch
[2] Ateleris GmbH, Brugg, Switzerland
simon.felix@ateleris.ch ivo.nussbaumer@ateleris.ch
filip.schramka@ateleris.ch

Abstract. We present an approach for using formal methods in embedded systems and its evaluation on a case study. In our approach, the developers describe the system in a restricted subset of the high-level programming language Scala. We then use 1) a verification system to formally prove properties of such Scala program, and 2) a source-to-source translator to map Scala to C code. We have adapted the Stainless verification system to support constructs for describing embedded software (more machine integer types and early returns) and to support verification patterns needed for embedded systems code (array swap operation, pre-allocated and initialized memory, constant-length arrays). The implemented C code translator generates code that can be compiled with compilers such as GCC and integrated into larger C applications.
We evaluate our approach on a case study of a file system of an instrument on the Solar Orbiter satellite. We have ported around a thousand lines of C code to Scala. We wrote specification and proof hints to make the code verify. Stainless verified the absence of run-time errors, as well as function preconditions, postconditions, and data structure invariants. The generated C code was integrated into the existing code base and exhibits very similar code size, memory use, and performance. In this process we identified multiple bugs in the well-tested code base, which were fixed in-orbit.

Keywords: formal verification · embedded system · file system · flight software · Scala · Stainless

1 Introduction

This paper includes our experience in using and adapting Stainless, a verifier for the Scala programming language [16], for a software component of a safety critical system. Safety-critical systems such as trains, cars, aircraft, satellites and

* Work financially supported by the Swiss Space Center project "Embedded Flight Software Verification (ESOVER)".

space probes contain embedded software that must be at all cost free of bugs. While extensive testing prevents many bugs, we aim to raise the correctness standard by additionally leveraging *formal verification* in the development process. With formal verification, we model the behavior of a program and prove that, under well-defined assumptions, the program behaves as expected in all executions.

Our use of Stainless is motivated by the knowledge of the tool by some of us, as well by our desire to make the experience appealing from both software development and verification experience point of view. Whereas our target application is a component of a custom file system, we choose to use a general-purpose tool, instead of a specialized one that may achieve higher automation [1,8,10,11,20], in part because we aim to arrive at conclusions and methodologies that generalize to other pieces of embedded software and because we wish to make assumptions of formal proof more explicit. Furthermore, formal verification tools may take several decades to become mature and develop a user community [18, 26, 28], an effort that is amortized over more use cases with general-purpose techniques. On the extreme end of this spectrum, interactive theorem provers have had long continued history of use and have high degree of trustworthiness. At the same time, they may appear unusual to developers accustomed to widely used programming languages. It is therefore natural to try and use a general-purpose and relatively mature verification tool while still remaining close to project source code. We found that Stainless enabled us to pursue this approach. While the original target of Stainless is (sequential) functional code, it has gained several features along the years, including support for imperative code. We show in this case study how Stainless can be used to verify real-world embedded code.

The experience that was driving our approach was formally verifying a portion of the file system of Spectrometer Telescope for Imaging X-rays, used onboard the Solar Orbiter satellite. We ported parts of the C code in this software to Scala and verified it using Stainless. Thanks to the use of Stainless, the resulting code was shown free from buffer and arithmetic overflows, two common problems in C. Furthermore, we also verified and proved additional properties, specified as invariants, preconditions, and postconditions.[3]

Using a Scala source to C source translator we incorporated into Stainless, we mapped the verified Scala code automatically to C, and used it as a drop-in replacement for the original C code in the existing system. Using this approach we were able to incrementally verify increasingly large components of the existing system, gradually replacing them with C code generated from verified Scala source.

Making this case study possible required us to add a new execution path to Stainless, which does not use Java Virtual Machine but C source code as target, without using memory allocation. Using C source code allowed us to use the gcc compiler available for a wide range of platforms, including LEON3 [14] soft

[3] A repository containing an illustrative fragment of the code we ported to Scala code and verified is https://github.com/epfl-lara/STIX-showcase.

core on which the software is deployed. Our translator generates readable code similar in structure to the Scala input.

A design choice of Stainless is to not use global variables but instead use the pattern of passing (possibly implicitly declared) parameters to functions, thus documenting program side effects. We have identified a combination of source code patterns (use of implicit parameters and initial values of default parameters) and code generation to respect this design choice. Consequently, we were able to support writing clean Scala code that can be mapped to the embedded code with statically allocated and initialized memory. To accommodate the use of unsigned machine integer types of various lengths, we extended the Scala front end of Stainless as well as our code generator to support these C data types and generate code that efficiently interacts with the surrounding embedded C code.

1.1 Contributions

This paper makes the following contributions:

- We present an extension of the Stainless verifier for handling embedded-style imperative code with statically allocated memory, fixed-sized arrays, early returns, and additional bitvector data types (Section 4).
- We show how to generate suitable embedded C code using source-to-source translation from Scala input to C code, extending a previous code generation approach of the Leon system [2] to recognize statically allocated memory use as well as to systematically eliminate specification-only (ghost) code (Section 5).
- As a case study, we present our experience in rewriting parts of the Spectrometer Telescope for Imaging X-rays (STIX) file system to Scala code, proving the absence of run-time errors, memory errors, as well as invariants, preconditions and postconditions. We have integrated the generated C code into the original project without loss of performance (Section 6).

1.2 Related Work

Cogent [1] is a high-level language specifically designed for formal verification of file systems. It features a compiler whose correctness is formally proven in the Isabelle proof assistant [27]. The authors of [1] wrote a file system and proved high-level properties. Other works strive for more automation using general techniques but tuned to file system models [8], or focus on finding bugs [20] instead of proving their absence. Interactive theorem provers have great expressive power for checking arbitrarily complex proofs, and they contain frameworks that help automate verification in various domains, including file systems [10, 11]. In contrast, in our approach, we write the specification in Scala, the same language (and in the same place) as the actual code. Dafny [24] has many similarities to Stainless; it was used in [9] to implement and verify operational crash-consistency file system models at a higher level of abstraction but was also used for low-level code in other projects [17].

SPARK Ada and other tools by AdaCore are alternative single-language options for high assurance software. Whereas Ada has the advantage of being designed for verification, it is not a functional language and does not support higher-order functions. We believe that functional programming is a strong basis for formal reasoning. That said, the approach of Stainless with preconditions and postconditions results in similar code in many cases, so it may even be possible to perform source-to-source translation between these two languages.

The Verified C Compiler VCC [4, 12] could be likely used to directly verify C code and has the advantage of supporting concurrency, though it also uses a different specification language than the implementation language. Before using Stainless to verify STIX code, a subset of authors tested CBMC [21] and Frama-C [13] on other parts of the code base. Both tools did not scale to the size of the code and struggled to work with the idioms in the application code and operating system. We suspect that both tools could have produced better results, had we invested more time. We suspect no tool will work out of the box entirely, even if it is designed to not require annotations as modular verification does. Because our Stainless-based verification approach results in C code, we could use tools like CBMC and Frama-C on the generated code to detect errors in the code generation step. In this project we rely on a code generation facility mapping a subset of Scala to C. Building such code generation implementation within a foundational framework such as CompCert [25] or CakeML [23] would further further improve the confidence in the resulting generated code.

2 STIX Instrument Onboard Solar Orbiter—Background

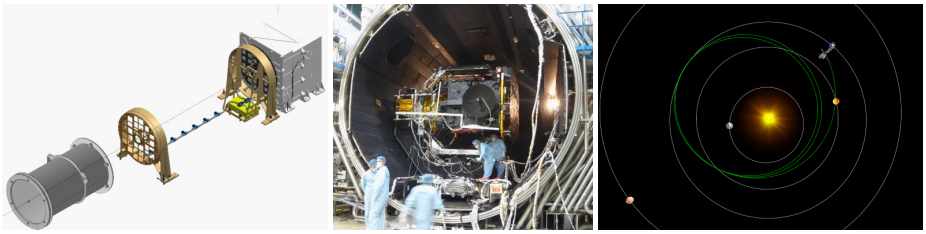

Fig. 1. *Left:* STIX images X-ray sources using moiré patterns produced by two tungsten grids placed in front of a sensor. *Center:* Solar Orbiter being prepared for launch. *Right:* Solar Orbiter completed its second Venus flyby maneuver November 2021.

The Spectrometer Telescope for Imaging X-rays (STIX [22]) onboard ESA's Solar Orbiter satellite is a hard X-ray imaging spectrometer. STIX observes hard X-ray bremsstrahlung emissions from solar flares and provides information about the hottest flare plasmas. The instrument and the satellite are shown in Figure 1. The satellite was launched in early 2020; the STIX instrument was turned on a few days later.

The STIX hardware consists of several custom application-specific integrated circuits (ASIC) and sensors, a radiation-hardened field-programmable gate array (FPGA), 128 MB DRAM, 2 MB SRAM, 1 MB EEPROM and 16 GB flash memory. The FPGA implements logic for real-time data processing, and a LEON3 [14] soft microprocessor. This SPARC V8-compatible soft microprocessor executes the flight software, which is the focus of this work. Owed to the limited energy budget and number of logic gates, the soft microprocessor runs at 20 MHz and is equipped with only 1 kB data and instruction caches. The system is under soft realtime constraints – missing interrupts means losing scientific data. The complete system processes up to 800'000 events per second and outputs a telemetry data stream of at most 700 bits per second.

The flight software is a self-contained C program, which is statically linked to the real-time operating system RTEMS [7]. To work around known bugs in the CPU a special, patched, GCC version is used to compile the software. The 36'418 non-comment code lines compile to 370 KB binary code. The flight software does not perform any dynamic memory allocation to prevent memory fragmentation. All data structures sizes are statically allocated at compile time. During development, several techniques were used to increase the robustness of the flight software: compiler warnings were enabled, static code analysis tools were run regularly, manual testing, automated end-to-end test scripts and unit-tests for certain subsystems were used.

Our verification efforts focused on the file system which manages the data stored on the 16 GB flash memory.

3 Background on Stainless Verifier

In this section, we highlight key features of Stainless verifier that we used to perform verification and, subsequently, code generation. Stainless was derived from Leon verification and synthesis system, which was originally designed to verify first-order recursive purely functional programs [30]. It was subsequently extended to support higher-order functions [31] and simple non-shared mutable data verified via a translation to functional code [5, 6]. Foundations and soundness of a substantial fragment of Stainless, including function termination, was presented using an expressive dependent type system, whose soundness is shown using a set-of-terms model [16]. When given Scala code, Stainless can process it in the *verification pipeline*. The typical deployment of Stainless programs (until the work in this paper) has been to compile them using Scala compiler and run on the Java Virtual Machine.

The verification pipeline of Stainless transforms high-level abstractions in the input program to simpler functional programming constructs which can be handled by our internal type-checker [16]. Our type-checker is not a typical type-checker in the sense that it not only ensures that "standard" types (such as `int`) are respected, but it also supports user-annotated assertions, and function pre- and postconditions in the form of boolean-typed expressions, which are encoded using *refinement types*.

The type-checker generates *verification conditions* for all annotations, which are formulas with recursive functions. All verification conditions must be checked to be true to ensure that assertions are indeed true for all possible function inputs respecting preconditions, and that function preconditions are respected at call-sites in all cases. In Stainless, verification conditions are checked using Inox [4], a solver for formulas written as functional programs with recursive functions, and which uses function unfolding [30] and SMT solvers (Z3 [15], CVC4 [3], Princess [29]) as backends.

4 Adapting the Verifier for Embedded Software

Despite the fact that Stainless was used to verify tens of thousands of lines of Scala code before, it was not suitable initially for verification of imperative embedded code.

4.1 Circumventing Stainless Aliasing Restrictions

When transforming away imperative features in the verification pipeline, Stainless checks that there is no *aliasing*, i.e. no two pointers to the same object. This greatly simplifies the transformation into a functional program, and therefore makes verification tractable for the solver.

The original file system code was written in a way that there could be several pointers to the same *control blocks* in the file system. Stainless would detect the aliasing and not transform the code. We made some adjustments to the STIX code ported to Scala in order to circumvent this restriction. Namely, all control blocks are stored in a global array, and wherever we needed to store a control block, we stored the index in the array instead. All control blocks accesses therefore go through the global array and there is no more aliasing.

4.2 Early Return Statements

The STIX code that we ported has early `return` statements in several places. We added a phase (`ReturnElimination`) in the verification pipeline to transform return statements into functional code. An often-used idea to translate imperative code into functional code is to use a form of continuation monad in order to know, at each point of the code (e.g. after a loop iteration), whether the code has already returned or not. To prove correctness in while loops containing return statements, we added the ability to specify a `noReturnInvariant`, which is an invariant that holds after each loop iteration except after a return.

[4] https://github.com/epfl-lara/inox

5 Scala to C translation for Embedded Software

To enable the deployment of embedded code, we incorporated the C code generator from the Leon system [2] into Stainless and used it as the starting point for our source-to-source generator. The code generation pipeline need not transform away imperative features into functional ones. For example, assignments and while loops remain mostly untranslated, as they can be directly mapped to their equivalents in C. The code generation pipeline shares some of the early phases with the verification pipeline, for example resolving method overrides and Scala class inheritance (`MethodLifting`). After that, we transform the program to an internal representation, where we perform some more transformations to produce a C program:

- `GhostElimination` removes all the ghost code specific verification,
- `Normalisation` flattens the block structure of a program, to avoid blocks within expressions (supported by Scala but not by C),
- `Referencing` adds references and dereferences where appropriate, as objects are passed by references in Scala, without explicit references,
- `IR2C` transforms classes to structs and enums.

In this section, we describe improvements we made in the C code generation pipeline [2] after porting it from the Leon system. These changes are what made it possible to write realistic components of the file system and generate C code with expected memory use and runtime behavior.

5.1 Unsigned Integers of Various Bit Lengths

The existing C code makes extensive use of several unsigned integer types (`uint8`, `uint16`, `uint32`), which were not supported by Stainless at the beginning of the project. The reason is that the Java Virtual Machine does not have support for native unsigned integers, and therefore, neither does Scala.

On the other hand, the used SMT backends support arbitrary-length bitvectors with signed and unsigned operations. We thus decided to add a Stainless library for signed and unsigned integers of arbitrary length (1 to 256), which is mapped in the verification pipeline to SMT bitvectors, and in the compilation pipeline to C signed/unsigned types, for bit lengths natively supported by C.

The library supports converting between signed and unsigned types, as well as narrowing and widening the bit length. These operations include appropriate checks (which can be locally or globally disabled) to detect overflows.

5.2 Mutable Global State

The verification pipeline of Stainless does not support verifying code with mutable global variables. We used a common Scala idiom to simulate global state: implicitly passing extra mutable objects functions that need to read or write the global state. We split the global state into several groups of mutable variables,

and each object has its own `case class` definition and corresponds to one such group. This has the benefit of explicitly showing in the function signature which parts, if any, of the global state are accessed by this function, and could be viewed as an effect system [19].

In the code generation pipeline, we remove these extra parameters from functions, and we leave three options to the user:

a. *(default)* Add a global declaration in the generated C code with a default value for each field of the case class. Additional annotations in Scala code, e.g. `static` or `volatile`, are carried over.
b. Add a global declaration in the generated C code without an initial value (implicitly zero-initialized), or
c. Do not add a global declaration. This is useful to refer to an existing variable declared in the existing C code, unknown to Stainless.

To ensure that this transformation is correct, we perform the following checks in the Scala code, for each case class `S` representing a global state portion. 1) Functions can take as argument at most one parameter of type `S`. 2) One function which does take such an argument is allowed to create instances of `S`, with default values, and pass it to other functions. 3) Instances of `S` can only be read, written to, or passed to other functions; instances cannot be copied or let-bound. These checks ensure that it is safe to remove parameters typed `S` and compile their read and write accesses to global C variables accesses.

5.3 Specifications and Ghost Elimination

We write the properties that we want to verify as preconditions (`require`), postconditions (`ensuring`), and code assertions (`assert`). Stainless is able to prove simple properties automatically, but more complex properties (e.g. sortedness of an array) require additional annotations in the form of:

a. functions to describe the property,
b. functions (lemmas) to prove that the property is maintained after an operation (e.g. insertion of an element in the array),
c. calls to these lemmas in the places where we need to prove the property.

During compilation, the preconditions (except in exported functions), postconditions, assertions, and additional annotations are eliminated in a *ghost elimination* phase. As such, they do not incur any performance overhead in the final executable.

In general, preconditions of exported functions are transformed into runtime assertions in C. For specific preconditions, the user can use the `require` keyword from `stainless.lang.StaticChecks` to denote that this precondition should not be compiled, even in an exported function. In general, this is unsafe as we do not know whether external function calls will respect these preconditions, but still useful for preconditions that may be too expensive to check at runtime (see one example Section 6.1), or preconditions that use Stainless features which are supported by the verification pipeline but not supported by the code generation pipeline.

5.4 Declarations Followed by `memset`

The following is a common idiom in C to initialize structures:

```
myStruct s;
memset(&s, 0, sizeof(s));
```

In Scala, this corresponds to declaring a variable `s` of (case) class `myStruct`, with all fields set to 0. When some fields have arrays, which themselves contains structs with other arrays inside, the mapping to C can become tricky. Therefore, when we encounter a zero-initialized case class that our C code generation pipeline does not support, we generate the idiom above instead.

In Scala, we can access array lengths, which we translate to structs containing a pointer, and an integer length (*bounded pointers*) in C. However, when an array is part of a struct, this makes the `memset` idiom above unusable, because `memset` would just set the pointer to 0 instead of setting the pointer to a preallocated memory region. In some cases, the length of arrays contained in a struct are known at compile-time, and we can compile them to fixed-length arrays, as shown in Figure 2.

```
case class MyStruct(ar: Array[Int]) {
  require(ar.length == 100)
}
```

```
typedef struct {
  int *underlying;
  int length;
} array_int;

typedef struct {
  array_int ar;
} MyStruct;
```

```
typedef struct {
  int ar[100];
} MyStruct;
```

Fig. 2. *Top:* a case class in Scala containing an array whose length is specified using a class invariant to be constant. *Left:* The generated C struct contains both a pointer and an array length when the class invariant is missing. *Right:* When a constant array length is specified as class invariant, the generated C struct contains a fixed-length array member instead.

5.5 Pure Functions

Because of the aliasing restrictions that we discussed in Section 4.1, Stainless contains an effect analysis that is able to determine which parts of the code mutate global state, and which parts are *pure*. We use this analysis during code generation to output purity annotations in the C code. Such annotations trigger additional optimizations in GCC, for example replacing deterministic function calls with constant values.

6 Experience with Case Study

We next present our experience in porting parts of the file system code from C to the subset of Scala supported by Stainless, and annotating it to prove the absence of run-time errors that Stainless always checks for, as well as proving additional invariants, preconditions, and postconditions.

6.1 Verified Properties and Statistics

The ported parts of the file system consist of around 6'000 lines of Scala code. This code contains 5'220 explicit and implicit verification conditions, all of which are proven (see Table 1). Initial verification takes 2'562 seconds[5], but verification completes in 86 seconds when using cached results from previous runs.

All of our data structures are array-based. Consequently, Stainless generates verification conditions for all array accesses and has to prove that all indices are within array bounds. To make verification of these bound checks feasible, we had to add invariants about the array lengths in function preconditions and in structures containing arrays, and we added invariants on integer indices in while loops. We show below a few examples of other higher-level properties we verified.

Table 1. Summary of the verification conditions.

Verification Condition	#	Verification Condition	#
Precondition	1546	Non-negative measure	57
Postcondition	1051	Strict arithmetic on shift	52
Array index within bounds	556	Measure decreases	19
Unsigned to signed overflow	518	Multiplication overflow	18
Class invariant	501	Division by zero	15
Subtraction overflow	284	Narrowing too large unsigned int	14
Addition overflow	246	Division overflow	6
Match exhaustiveness	128	Local invariant	5
Body assertion	124	Negation overflow	3
Signed to unsigned requires ≥ 0	74	Remainder by zero	3

Insertion into a Sorted Array The file system manages some data in a (fixed-length) sorted array. Insertion in this array uses an insertion sort that (1) looks for the index `i` where to insert an element by dichotomy, (2) shift all elements with lower priority to make place in the array, (3) assign the element to insert at index `i`.

As explained in section 4.1, the verification pipeline that we use for imperative code only supports limited forms of aliasing. Therefore, shifting mutable elements

[5] Measured on a MacBook Pro, Intel Core i9 2.3 GHz 8-Core, 32 GB RAM

in an array is not possible, because an assignment of the form `ar(i+1) = ar(i)` creates two aliases to the object initially stored in `ar(i)`. This problem led to the introduction of a new `swap(ar, i, i+1)` operation that swaps two mutable elements in an array without creating aliases. We were able to prove strong enough invariants in the while loops implementing the steps (1) and (2) above to show that the array remains sorted after insertion of new elements.

Counting Blocks with a Specific Status The flash memory managed by the file system is organized in blocks, each containing 256 kB data. During system initialization, each Flash block transitions from the initial state to one of the following states: *free*, *used*, *error*, or *bad*. Blocks in state *error* contain bit flips which are not correctable with the employed error correction codes. Those blocks can be reused for new data in the future. In extreme cases, the Flash hardware itself can fail due to aging or radiation. This leads to *bad* blocks, which should never be used anymore.

Instrument operators want to know how many blocks are in which state to assess the state of the flash memory. We store the number of blocks in each state in global counters. It is therefore natural to define an invariant that states that these global counters actually correspond to the number of control blocks with a specific status.

We defined the invariant using the recursive function `countStatus` that counts the blocks with a given status. Proving the invariant further required proving lemmas that explain how `countStatus` changes after updating the status of a block, which is not trivial given the recursive nature of *countStatus*. Specifically, it requires proving additional lemmas, which state the desired properties as postconditions, and which are themselves defined recursively following the `countStatus` pattern to simulate proofs by induction on the executions of `countStatus`.

6.2 General Improvement to Stainless

During the project, we continuously improved Stainless, either by fixing bugs, or implementing new features. In total, we merged around 150 pull requests related to this project in the public Stainless code base, and around 25 in the public code base of Inox, our backend solver.

To deal with a project this size we had to make performance improvements, for instance by supporting more recent backend SMT solvers (Z3 4.8.12 with its experimental "new core" option, CVC4 1.8), or by reducing the amount of duplication in the generated verification conditions.

To make solving of some verification conditions possible, we had to extend the `opaque` keyword to control at each call-site whether function bodies are visible to the solver[6]. Before, Stainless only supported the `opaque` keyword with per-function granularity.

[6] Thanks to Georg Stefan Schmid for an implementation idea of this feature.

6.3 Identified Bugs in the STIX File System Code

During this project we identified a number of implementation bugs in the existing file system code, of which we highlight two examples. First, we uncovered a potential buffer overflow due to an off-by-1 error in a data structure. The way the buffer was used prevented this problem from ever surfacing, but otherwise innocent changes might trigger the bug in the future, if left unfixed. Second, the type system of Scala helped identify a case where an incompatible `enum` type was returned by a function. Even though these bugs have no real-world ramifications, we patched the in-orbit instrument in December 2021.

6.4 Using Stainless without Prior Formal Verification Experience

Our team consists of experts that worked on the original file system implementation, and verification experts that were concerned with improvements to Stainless verification and code generation, as well as help in specification and verification.

Our experience with Stainless confirmed the expectation that formal verification of code is challenging without prior experience in the field.

First, it takes time to get accustomed to the language, in this case Scala subset supported by Stainless. For example, programmers cannot use the standard Scala class libraries or certain high-level abstractions, because they are unverified or rely on dynamic memory allocation. Instead, to write embedded code in Scala basic data structures must be implemented first. The resulting code is similar to the C implementation, but benefits from a richer type system. With these building blocks in place, we quickly adjusted to the way some constructs have to be expressed (e.g. enumerations, pass by reference, global variables).

A bigger challenge is specifying correct and verifiable properties. Some properties are straightforward to express or have proof obligations even generated automatically, like absence of arithmetic overflows or out of bound accesses. Other properties require recursive lemmas to encode inductive proofs in Stainless. The examples in Chapter 6.1 were only verifiable with assistance by the formal verification experts in the group.

6.5 Integration into the Existing C Code Base

In most cases, the generated C code can be integrated trivially in the existing C code base, because it has identical signatures. However, some concepts can be expressed in multiple ways in C. For example, the existing C code freely mixes arrays, raw pointers and bounded pointers, whereas the generated C code represents arrays as structs. Similarly, the existing code exploits the liberal C type system and preprocessor macros, which the generated code does not do. In such cases, it becomes necessary to convert between different representations at the interfaces. The required conversions are implemented as small, inlined functions with negligible overhead (Fig. 4). The call graph in Fig. 3 of the FSWrite function shows the STIX flight software and hardware drivers written in C, the file system in Scala, and how the bridge functions act as interfaces in-between.

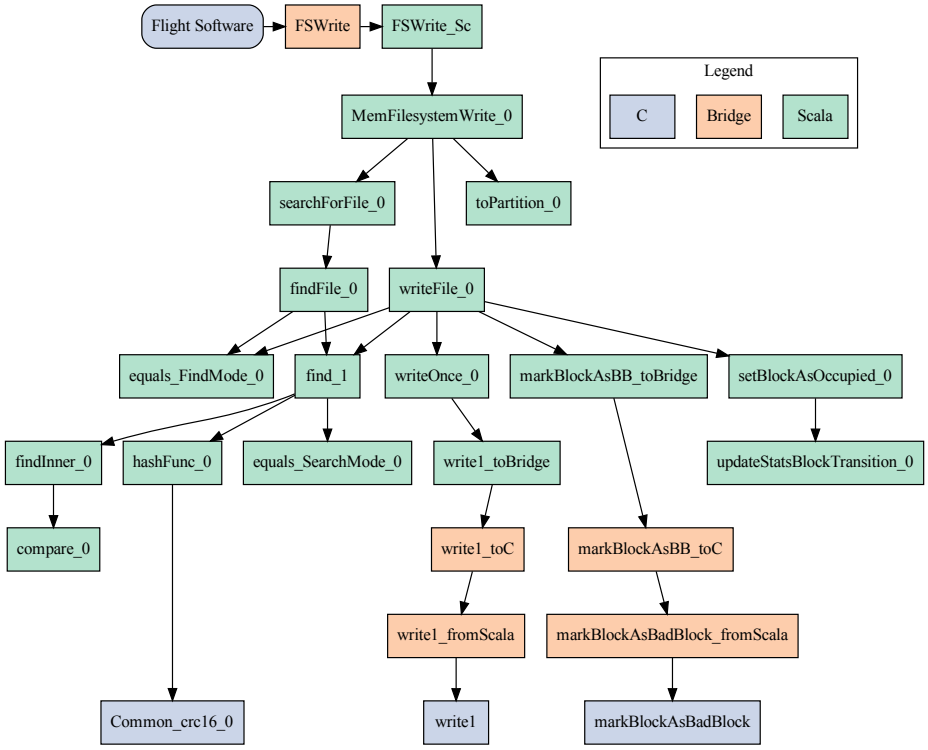

Fig. 3. Flight Software using the file system (top), and the hardware drivers (bottom) were not modified. Only the file system was ported to Scala. Bridge functions, written in C, connect the two implementations when function signatures differ.

6.6 Generated Code Performance, Memory, and Code Size Impact

In this case study we generate approximately 1 kLOC of C code from around 6 kLOC of Scala code (for implementation, specification and proof hints), which replaces a similar number of original C code. We compared the original flight software C code to the generated C code quantitatively and qualitatively. We focus our attention on file system metadata operations and microbenchmarks. The measurements were performed on an engineering model of the flight hardware. The engineering model contains 62'022 files in 7 partitions. During boot, the file system initialization code reads and processes all flash blocks. The next three tests operate on a particular file in the file system. A file is read, deleted, and finally written again. These operations perform a name-based lookup internally. Finally, we perform in-memory data microbenchmarks: endianness conversion and sample compression. It is important to note that we compare the generated

```
static array_uint8 toGenCArray(const void* x, int len) {
    return (array_uint8) { (uint8_t*)x, len };
}
void stream_write(MemStream_s* _s, void* _buf, uint32_t _bytes) {
    stream_write_scala(_s, toGenCArray(_buf, _bytes));
}
```

Fig. 4. Converting raw pointers to bounded arrays is trivial, due to the low level of C code. GCC optimizes these conversions, making it a zero-cost abstraction.

Table 2. Quantitative comparison between the original, hand-written C code and automatically generated C code. The reported sizes include the benchmark code. We report averaged results from 250 runs.

	Original C	Generated C	
Code size	513 072 bytes	514 368 bytes	(+0.3%)
Data size	21 824 bytes	21 744 bytes	(-0.4%)
Boot time	539 288 ms	560 305 ms	(+3.9%)
Read file (32 kb)	183 ms	176 ms	(-3.8%)
Write file (32 kb)	238 ms	242 ms	(+1.7%)
Delete file	5 ms	9 ms	(+55.6%)
Little-Endian decoding (224 kb)	404 ms	199 ms	(-50.7%)
Little-Endian encoding (224 kb)	797 ms	1006 ms	(+26.2%)
Compression (10^6 samples)	20 506 ms	20 566 ms	(+0.3%)

C code to a hand-tuned C implementation. The performance is comparable to the original C code for high-level operations (Table 2).

Significant increases in code size would not be acceptable: The CPU instruction cache has a limited capacity of only 256 instructions and significant performance drops occur when inner loops exceed this limit. Measurements confirm that the code and data sizes stayed almost identical. This is expected, as we carefully declared the data structures to correspond exactly to their existing C counterparts to ensure interoperability. The small data size reduction is caused by the replacement of a look-up table in the C version with an equivalent look-up function in Scala. The manual inspection of the resulting assembly code shows that GCC produces virtually identical outputs for inner loops in both cases.

We found that minor, innocent differences between the original and generated C code can have significant performance effects. For example, the extreme performance gaps observed in the Endian conversion microbenchmarks are not a result of major differences in the original and generated C code, but instead the result of different inlining decisions of the GCC compiler.

For quick operations, like file deletion, the performance overhead of bridge functions (see previous chapter) can become significant. However, the overhead is acceptable in the context of the overall system for our use case.

7 Conclusions

We have presented an approach for verifying embedded software implementations. The approach can be used to incrementally verify software by rewriting parts of it in a memory safe language and using source-to-source translation to produce C code that integrates into a large software ecosystem written in C.

To make this approach work, we needed to make substantial improvements to the original verifier, which was initially aimed at functional programs with memory allocated on the heap. We improved support for bitvector data types, including unsigned data types not present on the JVM. Furthermore, we added supported non-local **return** from functions, translating such code to compute a value of Either type (disjoint sum) encoding normal or early return outcome. We introduced new specification constructs for loops with such early returns.

A substantial change was to accommodate the use of global, statically allocated memory. We to preserved the design of Stainless where developers must use parameters to pass mutable parts of the heap and thus document function side effects. The design is convenient in Scala because function parameters can be declared implicit and omitted at the function call sites. To ensure that generated code uses only statically allocated memory that is appropriately initialized, we proposed a model that specifies initial values of fields of cases classes. Our code generator also recognizes data structure invariants that constrain Scala array sizes to be compile-time reducible to a constant; it maps such arrays to C arrays of constant size.

The executable Scala code we wrote in our case study has imperative flavor, so one may ask if the use of Scala and Stainless was justified. One reason is the use of a unified Scala notation. Indeed, even in imperative code, all control structures we used remain valid Scala; the language remains memory safe by design. Much of our case study is non-executable Scala code used to express preconditions, post-conditions and invariants. These specification (ghost) constructs widely use functional programming idioms with recursive functions and recursive data types. Ghost code never executes in the resulting system: Stainless proves it correct and eliminates it during code generation. The net result is that executable code is efficient, yet the developer has used constructs that belong to the same language for both code and specifications. This in contrast to verification systems where implementation and specification live in separate domains, which often results in unnecessary specification effort and a steeper learning curve for users.

Using our approach we verified components of the file system on the STIX instrument of the Solar Orbiter satellite. In this process we have identified and corrected several errors in the original system. We then established that the ported component of the code is free of run-time errors and that it satisfies basic invariants. The code size and performance of the generated code were on par with the original C code. We thus hope we presented a piece of evidence for feasibility of formal verification in embedded system domain.

References

1. Amani, S., Hixon, A., Chen, Z., Rizkallah, C., Chubb, P., O'Connor, L., Beeren, J., Nagashima, Y., Lim, J., Sewell, T., Tuong, J., Keller, G., Murray, T.C., Klein, G., Heiser, G.: Cogent: Verifying high-assurance file system implementations. In: Conte, T., Zhou, Y. (eds.) Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2016, Atlanta, GA, USA, April 2-6, 2016. pp. 175–188. ACM (2016). https://doi.org/10.1145/2872362.2872404
2. Antognini, M.: Extending Safe C Support In Leon. Master's thesis, EPFL (2017), http://infoscience.epfl.ch/record/227942
3. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: International Conference on Computer Aided Verification. pp. 171–177. Springer (2011)
4. Beckert, B., Moskal, M.: Deductive verification of system software in the verisoft XT project. Künstliche Intell. **24**(1), 57–61 (2010). https://doi.org/10.1007/s13218-010-0005-7
5. Blanc, R.W., Kneuss, E., Kuncak, V., Suter, P.: An overview of the Leon verification system: Verification by translation to recursive functions. In: Scala Workshop (2013)
6. Blanc, R.W.: Verification by Reduction to Functional Programs. Ph.D. thesis, EPFL, Lausanne (2017). https://doi.org/10.5075/epfl-thesis-7636, http://infoscience.epfl.ch/record/230242
7. Bloom, G., Sherrill, J.: Scheduling and thread management with RTEMS. ACM Sigbed Review **11**(1), 20–25 (2014)
8. Bornholt, J., Kaufmann, A., Li, J., Krishnamurthy, A., Torlak, E., Wang, X.: Specifying and checking file system crash-consistency models. In: Conte, T., Zhou, Y. (eds.) Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2016, Atlanta, GA, USA, April 2-6, 2016. pp. 83–98. ACM (2016). https://doi.org/10.1145/2872362.2872406
9. Bornholt, J., Kaufmann, A., Li, J., Krishnamurthy, A., Torlak, E., Wang, X.: Specifying and checking file system crash-consistency models. In: Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 83–98 (2016)
10. Chajed, T., Chen, H., Chlipala, A., Kaashoek, M.F., Zeldovich, N., Ziegler, D.: Certifying a file system using crash Hoare logic: Correctness in the presence of crashes. Commun. ACM **60**(4), 75–84 (mar 2017). https://doi.org/10.1145/3051092
11. Chajed, T., Tassarotti, J., Theng, M., Jung, R., Kaashoek, M.F., Zeldovich, N.: Gojournal: a verified, concurrent, crash-safe journaling system. In: Brown, A.D., Lorch, J.R. (eds.) 15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021. pp. 423–439. USENIX Association (2021), https://www.usenix.org/conference/osdi21/presentation/chajed
12. Cohen, E., Dahlweid, M., Hillebrand, M.A., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5674, pp. 23–42. Springer (2009). https://doi.org/10.1007/978-3-642-03359-9_2

13. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C. In: International conference on software engineering and formal methods. pp. 233–247. Springer (2012)
14. Daněk, M., Kafka, L., Kohout, L., Sỳkora, J., Bartosiński, R.: The LEON3 processor. In: UTLEON3: Exploring Fine-Grain Multi-Threading in FPGAs, pp. 9–14. Springer (2013)
15. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)
16. Hamza, J., Voirol, N., Kunčak, V.: System FR: Formalized foundations for the Stainless verifier. Proc. ACM Program. Lang. **3**(OOPSLA) (Oct 2019). https://doi.org/10.1145/3360592, https://doi.org/10.1145/3360592
17. Hawblitzel, C., Howell, J., Lorch, J.R., Narayan, A., Parno, B., Zhang, D., Zill, B.: Ironclad apps: End-to-End security via automated Full-System verification. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). pp. 165–181. USENIX Association, Broomfield, CO (Oct 2014), https://www.usenix.org/conference/osdi14/technical-sessions/presentation/hawblitzel
18. Inria, C., contributors: Early history of coq. https://coq.inria.fr/refman/history.html (2021)
19. Jouvelot, P., Gifford, D.K.: Algebraic reconstruction of types and effects. In: Wise, D.S. (ed.) Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida, USA, January 21-23, 1991. pp. 303–310. ACM Press (1991). https://doi.org/10.1145/99583.99623
20. Kim, S., Xu, M., Kashyap, S., Yoon, J., Xu, W., Kim, T.: Finding bugs in file systems with an extensible fuzzing framework. ACM Trans. Storage **16**(2), 10:1–10:35 (2020). https://doi.org/10.1145/3391202, https://doi.org/10.1145/3391202
21. Kroening, D., Tautschnig, M.: CBMC–c bounded model checker. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 389–391. Springer (2014)
22. Krucker, S., Hurford, G.J., Grimm, O., Kögl, S., Gröbelbauer, H.P., Etesi, L., Casadei, D., Csillaghy, A., Benz, A.O., Arnold, N.G., et al.: The spectrometer/telescope for imaging X-rays (STIX). Astronomy & Astrophysics **642**, A15 (2020)
23. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: Cakeml: a verified implementation of ML. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014. pp. 179–192. ACM (2014). https://doi.org/10.1145/2535838.2535841
24. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: International Conference on Logic for Programming Artificial Intelligence and Reasoning. pp. 348–370. Springer (2010)
25. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009). https://doi.org/10.1145/1538788.1538814
26. Moore, J.S.: Milestones from the Pure Lisp theorem prover to ACL2. Formal Aspects Comput. **31**(6), 699–732 (2019). https://doi.org/10.1007/s00165-019-00490-3
27. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: a proof assistant for higher-order logic, vol. 2283. Springer Science & Business Media (2002)
28. Paulson, L.C., Nipkow, T., Wenzel, M.: From LCF to Isabelle/HOL. Formal Aspects Comput. **31**(6), 675–698 (2019). https://doi.org/10.1007/s00165-019-00492-1
29. Rümmer, P.: A constraint sequent calculus for first-order logic with linear integer arithmetic. In: Proceedings, 15th International Conference on Logic for Program-

ming, Artificial Intelligence and Reasoning. LNCS, vol. 5330, pp. 274–289. Springer (2008)
30. Suter, P., Köksal, A.S., Kuncak, V.: Satisfiability modulo recursive programs. In: Static Analysis Symposium (SAS) (2011)
31. Voirol, N., Kneuss, E., Kuncak, V.: Counter-example complete verification for higher-order functions. In: Scala Symposium (2015)