Design and Implementation of a Verified Interpreter for Additive Manufacturing Programs (Experience Report)

Matthew Sottile
Lawrence Livermore National Laboratory
Livermore, CA, USA
sottile2@llnl.gov

Mohit Tekriwal
Lawrence Livermore National Laboratory
Livermore, CA, USA
tekriwal1@llnl.gov

Abstract

This paper describes the design of a verified tool for analyzing tool paths defined in the RS-274 language for 3D printing systems. We describe how the analyzer was designed to allow a mixture of verification and code-extraction techniques to be combined for constructing a correct toolpath analyzer written in the OCaml language. We show how we moved from a fully hand-written OCaml program to one incorporating verified components, highlighting architectural decisions that were made to facilitate this process. Finally, we share a set of architectural lessons that are generally applicable to other software with a similar goal of integration of verified components.

CCS Concepts: • **Theory of computation → Program verification**.

Keywords: RS-274 language, formal verification, 3D printing, functional programming

ACM Reference Format:
Matthew Sottile and Mohit Tekriwal. 2024. Design and Implementation of a Verified Interpreter for Additive Manufacturing Programs (Experience Report). In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Software Architecture (FUNARCH '24), September 6, 2024, Milan, Italy*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3677998.3678221

1 Introduction

Verification techniques are used when building software where we require strong evidence that it is correct. Integration of verified and unverified components can be complex due to the need to isolate data and logic to aid in the specification and proof process. In this paper we discuss the design, implementation, and verification of a prototype OCaml-based tool for analyzing 3D printer programs written in the special purpose RS-274 ("G-code") language[5]. This interpreter is used to assess whether the 3D printer code and the resulting physical artifact meet requirements that would affect the quality of the object being manufactured. The verification of the tool implementation itself provides a strong foundation of trust for the analysis it performs on its inputs.

In 3D printing there are two common approaches to verification: simulation with visualization, and physical examination of the artifact after fabrication via destructive or non-destructive testing [7]. No prior work has been found that approaches the problem in a manner to software verification. The goal of verification is to detect whether the command sequence that drives the printer introduces subtle defects in the object being fabricated (such as misaligned extrusion or incorrect dimensions). Our approach is based on a direct interpretation of the G-code where we evolve a model of the machine and object being printed such that verification conditions can be checked at each step of the print. This ability to perform complex checks against the physical state model is unique to our tool. Furthermore, the design of the tool allows them to be added without compromising the correctness of the overall interpreter loop that is responsible for modeling the generic semantics of G-code programs.

The initial prototype of this tool was written in pure OCaml with a light set of external dependencies. The choice to minimize the dependencies of the tool was in anticipation of the verification process that we planned to apply: complex third party dependencies complicate verification as we have little control over their implementation. Once we created the initial prototype and satisfied ourselves that it was correct on the basis of testing, we moved to the second phase of the project: verification of as much of the interpreter as possible using formal verification techniques. In this paper we describe how we adapted the architecture of the tool to allow for verification of components and code extraction from verified specifications to obtain executable code. This extracted code replaced hand written code such that our implementation not only passed tests, but was accompanied by verified evidence of conformance to formal specifications.

FUNARCH '24, September 6, 2024, Milan, Italy

ACM ISBN 979-8-4007-1101-5/24/09
https://doi.org/10.1145/3677998.3678221

In Section 3 we provide an overview of the interpreter design and explain the analysis that it performs on input G-code. In Section 3.1 we discuss the verification approach and architectural changes that we made to allow for formalization and subsequent verification. We describe our use of two different verification techniques: deductive verification via satisfiability modulo theories solvers with Why3, and formalization in a traditional proof assistant using Coq and the Flocq floating point theory. In Section 4 we briefly discuss the results of the verification process and the integration of hand written and extracted code. Finally, in Section 5 we describe a set of architectural lessons that we learned in this process that we believe represent general concepts that can be applied to the construction of similar tools in the future.

The work described in this paper is based on what is achievable using the current state of verification tools like Coq and Why3 as of the date of publication. Our long term goal is to achieve the same level of verification with as much automation as possible. As Why3 improves and adds support for richer floating point features we expect to revisit applying it in places where current versions did not meet our requirements. We are actively exploring ways to reduce the manual proof burden as we expect this G-code analysis tool to continue to evolve as we add additional verification conditions, support for vendor extensions to G-code, tune performance, and so on. The burden of manual proof construction in Coq currently slows down development, but is a tradeoff necessary to achieve our goals of tool verification.

1.1 Broader Motivation

When we consider the workflow for an additive manufacturing process the steps typically consist of a design process, a toolpath generation process, and then execution of the toolpath on the fabrication device. In the design phase that uses traditional computer aided design tools, and potentially modeling and simulation tools, we can establish a trust case using traditional software verification and validation techniques [2, 8]. Similarly, the toolpath planning phase that maps a design (e.g., a constructive solid geometry model) to a sequence of tool actions, we again can perform verification of the toolpath planner itself using traditional methods. The device itself that typically has some firmware and system software is again an instance of traditional software, which allows us to verify its correctness. The gap that we have is the program that represents the toolpath itself. If the toolpath planning software and device firmware are opaque (e.g., proprietary software), we cannot analyze them directly to determine if the toolpath that is produced will conform to the requirements of the user.

Instead, we must resort to an analysis that is similar to binary analysis of regular programs: when we cannot verify that a compiler is correct we instead analyze the output of the compiler against a semantic model of the machine that will execute it. In the case of this work, we consider the toolpath planner to be the "compiler" of the design, and the RS-274 standard to define the semantics of the tool path language along with constraints imposed by the specific target printer (such as physical constraints). Much like verification of binaries on commercial CPUs, we are limited in terms of access to the firmware of the device to obtain the semantics and must carefully define them ourself within our analysis tool.

2 Overview of the RS-274 Standard

The Intelligent Systems Division of the National Institute of Standards and Technology (NIST) provides a standard containing an informal description [5] of the RS-274/NGC programming language used in Numerical Computing (NC) machine tools. Although this standard was initially proposed for traditional machining tools like lathe, milling, drilling, they have been adopted as the command language for many 3D printers. The RS-274/NGC language (often referred to as G-code) is based on a line-oriented execution model. Each line may include commands to a machine to perform a task, like move in a straight line while extruding filament during 3D printing, move in a circular arc, or do a rapid reset of the head of the machining tool. The words for these commands are called "G codes" and "M codes". G codes mostly operate on the physical state of a machining tool, like linear motion, directed circular motion, setting the coordinate system data, plane select, feed rate and length of the extruded filament. M codes mostly operate on the controller state like program stop and end, coolant on and off, extrusion mode select, override control, and so on.

In RS-274/NGC, many commands cause a machining center to change from one mode (state) to another. Some of these modes stay active until some other command changes it explicitly. Such commands are called "modal" commands, like the G90 command, which affects the distance mode. If a G90 command is specified in the NGC program, the machining tool moves in a absolute coordinate system relative to a fixed origin. If a G91 command is specified later in the NGC program, the program will switch modes and subsequent motion will be relative to the current tool position.

Other commands available are "non-modal": their effect is restricted to the line in which they occur and do not persist into subsequent code lines. For example, the G28 command to return to a home position is executed only on the line where it occurs. Similar non-modal commands exist for one-time changes to settings such as the coordinate system origin.

The modal commands are arranged in sets called "modal groups". The standard specifies that only one member of a modal group may remain active at any given time. For example, the motion commands G0 for rapid motion, G1 for linear motion, G2 for clockwise arc motion and G3 for anticlockwise arc motion are part of the modal group 1. At any given time, only one of them will remain active. If two modal

commands in the same group are specified at any given time, that should result in an error. We model each modal group as a state transition system that defines legal states that the group can be in and legal transitions that can occur.

Modal groups are independent from each other: transitions in a modal group will not affect the state of any other group. For example, the distance mode group (G90, G91) is independent from the motion mode group (G1, G2, etc.). For several modal groups a default state must be selected when the machine is ready to accept commands. The distance mode for example is necessary before any motion command can be interpreted as distance mode is required to calculate position information. This led to the following encoding of modal groups in our Coq model:

```
Record modal_group := mkModal {
  values : list string;
  default : option string;
  required : bool; }.
```

For example, the distance mode group (group 3) would have `values` of G90 and G91, a `default` of G90, and the `required` flag set to true. It should be noted that the data structure that is modeled directly reflects the configuration file that a user specified which is designed to be very simple and avoid conditional formatting. For example, modal groups that are required (i.e., they must always have a value) must have a default provided, but those that are optional may specify no default. Instead of allowing the presence of the default field to vary across groups, we opted for uniformity of fields resulting in some redundant information. Revisiting the configuration to streamline it may be revisited in the future.

3 Approach or Design of the Tool

Given the definition of RS-274 in Section 2, we designed our interpreter to model the controller state as well as the state of the physical object being manufactured. We are specifically focused on the portions of G-code relevant for additive manufacturing systems and therefore omit commands related to other kinds of machines (e.g., lathes or CNC machines).

The interpreter maintains two state models: the controller state and the physical state. The controller state holds information such as modal states (e.g., extrusion state, feedrate, etc.) and non-modal states (parameters that are consumed by G or M-codes). The physical state model maintains the configuration of the manufacturing device (e.g., extrusion nozzle or cutter position) and the history of actions performed so far, such as the position and size of extruded material. This history of actions is sufficient to construct a model of the physical object being printed. We maintain the full history of actions instead of just a model of the object so we can capture actions where the physical action affected the machine but not the object (e.g., nozzle motion while extrusion is disabled).

G-code is parsed into a parse tree that is traversed and analyzed to create an abstract syntax tree that captures the semantics of each line of G-code. The G-code language defines a priority order for evaluating commands that may be different than the order that they appear in a single line. This requires the parse tree to be analyzed on a line-by-line basis to map each parsed line into a normalized form (the abstract syntax tree) that reflects the execution order as defined by the G-code standard.

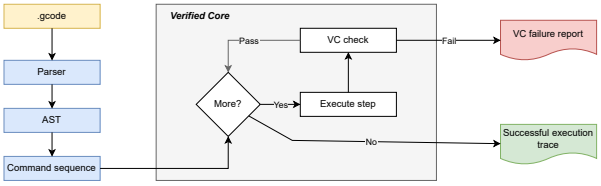

Figure 1. High-level interpreter design indicating scope of verified components.

The overall structure of the tool follows the flow shown in Figure 1, which is a typical interpreter design. Given a sequence of lines with the commands on each line put in appropriate evaluation and parameter order, the interpreter steps through each instruction by clearing any non-modal states and passing the overall interpreter state (both physical and controller state) to the appropriate command handler to produce the updated state at the next step. This process repeats until the line sequence has been exhausted and the resulting physical state (including its evolution history) can be emitted. During this interpreter loop, the state is passed through a verification condition checker that enforces domain-specific constraints on the interpretation. For example, the simplest physical constraint is that the tool stays within a physical bounding box. If the tool leaves the bounding box, a violation is reported. The tool is designed to allow other more complex verification condition checks to be integrated without impacting the rest of the interpreter.

3.1 Design Considerations

Given our initial prototype of the tool we identified a number of barriers to verification that required us to reevaluate some of our design choices. First, we determined that some components of the tool would not be possible to verify in a reasonable amount of time: I/O and parsing facilities in particular. I/O is notoriously difficult to model, and the RS-274 standard does not provide a formal grammar of G-code programs leading to a lack of specification against which a parser could be evaluated. We determined that these components at the "edge" of the tool were not critical and could be tolerated for the time being in the trusted computing base.

The second barrier we encountered was due to limitations of theories supported by automated tools. We chose the Why3 [3] tool that allows for semi-automated verification of

specifications written in WhyML. Why3 is appealing because it automates the process of proof generation and verification by external SMT solvers, and allows one to experiment with different solvers that may have different performance or capabilities. Why3 also supports code extraction, allowing us to obtain OCaml code from specifications once they are verified to conform to their requirements.

Unfortunately, Why3 and WhyML are limited in their support for floating point arithmetic used by our physical model and we encountered limitations in their support for OCaml-style functorized maps. The limitations related to floating point code caused us to seek an alternative mechanism for formalizing the physical models, and the limitations related to data structures caused us to redesign our internal structures to use simpler (but less performant) implementations in the interest of verifiability. Interestingly, while our choices to replace data structures moved to a design that has worse asymptotic time complexity, we discovered that in practice the difference was negligible - thus the choice to opt for verifiability outweighted any benefits of better asymptotic performance. This is due to the use of maps to represent the modal states and configuration of the interpreter, both of which are finite and relatively small (on the order of tens of elements).

To overcome the floating point issue, we redesigned the internal flow and state model of the interpreter to cleanly separate the logical controller state from the physical model state. In doing so, we were able to formalize the physical model state and corresponding numerical algorithms in Coq using the Flocq [1] floating point theory. This reduced the automation of our verification process as compared to the Why3 approach, but allowed us to verify properties not currently possible via Why3. We were also able to use the code extraction facility within Coq to obtain OCaml code similar to that extracted from Why3.

Our final step was to formalize the overall interpreter loop and verification condition checker. Both of these we chose to formalize in Coq. The most important reason was that most verification conditions that we wish to check are applied to the physical state (e.g., the state of the machine or the state of the object being fabricated). Therefore we require reasoning about floating point numerical operations which we already encode in Coq for verifying the numerical components of the core interpreter. An additional consideration that was convenient was the observation that it was easier to use verified components from Why3 as part of the proof process in Coq than vice versa.

3.2 Formalization and Extraction

The verified components of the tool are broken down in Figure 2 to show which parts have been formalized in Why3 and Coq, as well as their dependency relationships. This verified base consists of code extracted from the implementation of data structures verified in Why3 and the linear algebra routines extracted from Coq.

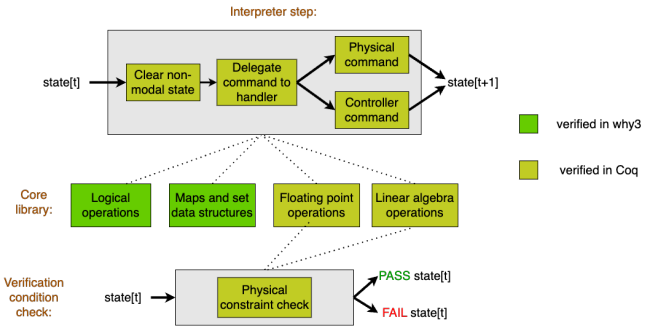

Figure 2. Illustration of the verification effort in Coq and Why3 and dependency graph of the verified core

The main property we have verified about the interpreter is the correctness of interpreter step function, which states that if certain physical and modal invariants are preserved on the input state to the interpreter step function, and some pre-conditions are respected on the machine configuration, parameter handling and on the AST, then those physical and modal invariants are also preserved for the output state, assuming there is no error in the intermediate operations. If an error occurs at any of the intermediate operations in modifying the physical and modal states, then those errors are propagated throughout, and the interpreter step function results in an error state. An example of a physical state invariant, denoted as `physical_state_invariant` in later parts of the paper, is the finiteness of the position vector of the printer head or the finiteness of the extrude position for the filament, relative to the printer head. Since, the modal state maps a modal command like "motion" to its corresponding value, the modal state invariant, denoted as `modal_state_invariant`, states that every key in this map is unique, i.e., there is a one-to-one map between a modal command and its value at any point in the program execution.

3.2.1 Why3 Verification and Extraction.

We illustrate the use of Why3 to take a specification of a simple string set membership function, verify that desired properties hold, and then extract an OCaml implementation. The WhyML specification of this operation is defined as follows:

```
let rec string_set_mem (x : string) (l : list string) =
variant {l}
ensures {Mem.mem x l = result}
    match l with
    | Nil -> False
    | Cons y r -> x = y || string_set_mem x r
    end
```

The WhyML implementation `string_set_mem` is verified using Hoare-style program reasoning. Given a key `x` and a list `l`, we verify that the `string_set_mem` program terminates using the `variant` clause and the result from this program

is same as one obtained from a mathematical definition of membership for a list (via the Why3 definition of `Mem.mem`). When Why3 successfully discharges the proof obligations for this definition, we then use the OCaml extraction driver provided by the Why3 developers to extract a verified implementation to OCaml as follows:

```
let rec string_set_mem (x: string) (l: (string) list) : bool =
  match l with
  | [] -> false
  | y :: r -> x = y || string_set_mem x r
```

3.2.2 Coq Verification and Extraction. We used Coq for aspects of the code that work with floating point arithmetic, allowing us to define the operation, prove relevant properties, and extract an OCaml implementation.

As noted earlier, a main reason for our shift to Coq for modeling the physical state and the controller physical state was inadequate support for floating-point operations in Why3. Specifically, we had issues defining floating-point literals in Why3. For instance, consider the following whyML code for computing the average of two floating-point values

```
use ieee_float.Float64
let sum (a : Float64.t) (b: Float64.t)= (a .+ b)
```

When we use the `ocaml64` and our custom floating-point driver to extract an OCaml implementation from the above WhyML implementation using the prompt

```
 why3 extract -D ocaml64 -D map_float.drv
test_float_literal.mlw -o test_float_literal.ml
```

we get the following OCaml code:

```
  let sum (a: float) (b: float) : float = (a +. b)
```

However, when we compute the average using the following code in WhyML

```
  use ieee_float.Float64
  let sum (a : Float64.t) (b: Float64.t) = (a .+ b)
  let const = (2.0 : Float64.t)
  let average (a : Float64.t) (b: Float64.t) =
      (sum a b) ./ const
```

and run the extraction command:

```
  why3 extract -D ocaml64 -D map_float.drv
   test_float_literal.mlw -o test_float_literal.ml
```

we get the following error:

```
  anomaly: File "src/extract/compile.ml", line 259,
           characters 17-23: Assertion failed
```

This error seems to be with the lack of support for extraction of floating-point literals in the source file of Why3. On our correspondence with the Why3 developers in the Why3 zulip chat, they confirmed this issue with the floating-point literals for the latest version of Why3, we were using at the time of development. While Why3 allows reasoning on floating-point literals, the issue with floating-point literals seems to be only with the code extraction. Since we rely heavily on extraction of floating-point literals in our analysis tool, we had to shift from Why3 to Coq. That said, the Why3 developers did assure us that this issue will be handled in the upcoming versions of Why3. Since one of our main goals in the design of tool was to automate the reasoning as much as possible, we would consider re-writing our tool in whyML for verifying the physical properties of our tool in Why3.

An example of an operation on floating-point vectors defined in Coq is as follows:

```
Definition vec_add (a: vec3) (b : vec3) := mkvec3
(add a.(x) b.(x)) (add a.(y) b.(y)) (add a.(z) b.(z)).
```

Here, `vec3` is a vector type defined over primitive floats in Coq. The `add` operation is addition of two primitive floats. We write a contract for this vector addition operation in the Hoare logic style in the lemma `vec_add_is_finite` with the pre-conditions being the finiteness of each element in the vector and absence of overflow in addition of two float values. The post-condition specifies that each element in the resulting vector is finite.

```
Lemma vec_add_is_finite (a b: vec3):
is_finite a.(x) =true /\ is_finite a.(y) =true /\
is_finite a.(z) =true /\ is_finite b.(x) =true /\
is_finite b.(y) =true /\ is_finite b.(z) =true ->
sum_no_overflow (B2R (Prim2B a.(x))) (B2R (Prim2B b.(x))) ->
sum_no_overflow (B2R (Prim2B a.(y))) (B2R (Prim2B b.(y))) ->
sum_no_overflow (B2R (Prim2B a.(z))) (B2R (Prim2B b.(z))) ->
is_finite (add a.(x) b.(x)) = true /\
is_finite (add a.(y) b.(y)) = true /\
is_finite (add a.(z) b.(z)) = true.
```

Here, we use the primitive floats for implementing floating-point operations because of their support in Coq's extraction mechanism. We interface with the Flocq's [1] floating-point theory for the correctness proof.

We follow a similar approach for defining and verifying all of the required data structures and vector operations. As mentioned earlier, one of the key challenges in our design was a careful de-coupling of physical and controller states, which is illustrated using the following Coq definitions:

```
Record physical_state :=
mkphysical_state {
  time : float;
  head : vec3;
  relhead : vec3;
  extrude_pos : float;
  extrude_on : bool;
  history : list path
}.
```

```
Record controller_state :=
mkCS {
  c_physical :
  controller_state_physical;
  c_modal    :
  controller_state_modal;
 variables : string_map ast
}.
```

This nice de-coupling made the verification of some key invariants easy. For instance, consider the following instantiated theorem statement, which verifies the correctness of the interpretation of radius in the interpreter AST.

```
modal_state_invariant st.(controller).(c_modal) ->
controller_state_constraints st.(controller).(c_physical) ->
physical_state_invariant st.(physical) ->
(exists st' : machine_configuration,
   Result (interp_radius_update st a) = Result st' /\
   modal_state_invariant st'.(controller).(c_modal) /\
  controller_state_constraints st'.(controller).(c_physical)
   /\ physical_state_invariant st'.(physical) ) \/
(exists msg : string,
Result (interp_radius_update st a) = Fail msg)
```

where `modal_state_invariant` defines some invariants that need to be satisfied for the purely modal state of the controller, `c_modal`. `controller_state_constraints` defines some invariants that need to be satisfied for the controller physical state, `physical_state_constraints` and `c_physical` defines the physical state of the machine. Since, `radius` is a field in the `c_physical` sub-field of type `controller_state_physical`, defined as

```
Record controller_state_physical := mkCph {
  feedrate : float;
  axes_parameter : axis_state;
  radius  : option float;
  coordinate_offset : vec3;
  extrusion_offset : float
}.
```

it just modifies the controller physical state, without affecting the controller modal and the machine physical state, i.e., $st'.(controller).(c\_modal) = st.(controller).(c\_modal)$ and $st'.(physical) = st.(physical)$. We can thus plug these equalities in the goal, and use the preconditions on the modal state and physical state invariants on the input state st to prove invariants on the output state st', trivially. Therefore, the only invariant that needs to be proven after an interpretation of the radius is the `controller_state_constraints`. We prove this using the following lemma statement

```
Lemma interp_radius_correct:
 forall (cfg : config) (st : machine_configuration) (a : ast),
 is_finite (v_interp_float a) = true ->
 controller_state_constraints st.(controller).(c_physical)
 -> exists st' : machine_configuration,
 controller_state_constraints st'.(controller).(c_physical)
 /\ interp_radius_update st a = st'.
```

3.2.3 Integration of Verified Why3 and Coq Components.

As illustrated, we further de-couple the controller-state into physical and purely modal controller states. Properties of the purely modal controller states are verified in Why3, and properties of all the physical states including the controller physical states are verified in Coq. There is however some dependence of the physical model on the purely modal models, which poses challenge in a clean de-coupling. To overcome this issue, we use a faithful translation of the extracted OCaml from WhyML to Coq using the coq-of-ocaml tool [6] (which becomes part of our trusted computing base.) However, this share of the TCB is small and by proving the soundness of the interpreter in Coq, we ensure that this share of the TCB is reliable.

We then compose the physical and controller states into a `machine_configuration type`, defined in Coq as follows:

```
Record machine_configuration := mkMC {
  physical : physical_state;
  controller : controller_state
}.
```

This decomposition of the machine state to physical and controller state allows us to reason about each machine operation modularly and scalably. This decomposition also allows us to leverage the automated verifiers accessible via Why3 as much as possible, thereby reducing the number of manual proofs that we need to perform for verifying the system.

4 Results

The primary focus in the design of this tool has been to minimize the trusted computing base and extract the verified core of the interpreter. To that end, 46% of the total code base is handwritten, which includes the G-code parser, JSON parser for configuration files, I/O functions, AST generator and pretty printers for the output from the interpreter. The following table compares extracted lines of code with handwritten code.

Extracted code from Coq	3376 LoC
Extracted code from WhyML	139 LoC
Handwritten code	3000 LoC

Out of the total extracted code base, all the utility functions like string maps and sets were verified in Why3. We used two SMT solvers: Alt-ergo and z3. Most of the simpler proof obligations were discharged with these SMT solvers without manual intervention. However, some of the complicated proof obligations like `string_map_add`, which involves composition of helper functions, required some manual intervention to simplify the proof obligations and transform it into a form which can eventually be automatically discharged using these SMT solvers.

These utility functions have been used in the parser, ASTs, driver functions and in the definition of modal updates. The linear algebra components were formalized in Coq using Flocq's floating-point theory and extracted to OCaml. These linear algebra routines were used to define the kinematics and geometric computations used by the interpreter in Coq. These numerical primitives are also used by the verification condition checker within the tool to check the physical state model after each interpreter step.

The verification checker currently implements a simple check that the nozzle stays within the bounding box for linear and arc motion steps. Arc motion is more complex to check than linear motion since it is possible that for a given arc radius, the end points of the arc may reside within the box but the arc may leave it. Linear motion guarantees that if the end points are in the box, every point in between will be as well. Any violation in these conditions is immediately detected and reported. Figure 3 illustrates the violation detection from our verification checker. We are also working on writing a verification checker to ensure that the printer nozzle does not collide with the printed object.

The existing checkers were chosen as they can reuse the verified numerical algorithms that are used to evolve the physical model of the interpreter. Our plans for more sophisticated checks that require algorithms and data structures beyond those already present in other parts of the interpreter, such as collision detection, will require additional work for specification and verification. We have started this process

using a similar approach as the rest of the tool: we have built a pure OCaml implementation of a collision detection method for testing and evaluation. The specific verification conditions we are interested in using this will include determining if the tool collides with the print-in-progress or the geometry of the printer itself. In addition to collision detection as the supporting algorithm, we are studying appropriate ways of representing the printed object such that it represents enough physical detail to accurately detect issues. Once we have demonstrated that our approach to implementing this class of checkers is appropriate and suffficiently efficient for practical use, we plan to replace as much of it as possible with code extracted from a verified implementation in a system like Coq. The choice of Coq in this case is largely based on a desire to build upon recent related work in this area that presents a verified k-D tree structure [4].

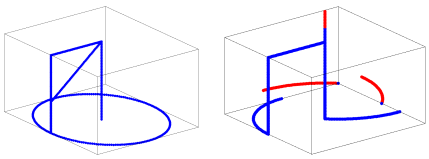

Figure 3. Detection of a violation of the bounding box verification condition. Red traces indicate portions of the tool path that violate the condition, in this case caused by introducing spurious motion mode changes in the command sequence.

5 Architectural Lessons

Our work demonstrates integration of verification approaches with the design of a scalable analyzer for high-precision additive manufacturing. The key architectural lessons from our design of a verified analyzer are summarized as follows.

5.1 Modularity and Careful Choice of Data Structures Amenable to Verification

This correct by construction framework was achieved by carefully decomposing a large piece of software into small units, which can be verified individually and modularly composed to obtain a verified tool. To achieve our goal of a verified G-code analyzer, we had to restructure our initial prototype, which was implemented in OCaml and had a tight coupling between the physical and controller states of a 3D printer. This coupling made the verification process tricky, partly due to the limitations of the existing verification tools: specifically, the limited support for floating point code in Why3. Our design philosophy for a scalable verified G-code analyzer was to automate the verification task and abstract away the components of the analyzer for such automated verification as much as possible, which can then be re-used for similar problems.

This preference for automation motivated us to use the Why3 [3] tool, which integrates the SMT solvers and interactive theorem provers, allowing us to offload much of the verification tasks to automated solvers. This enabled us to discharge correctness proofs about the data-structures used in our analyzer, using a push-button approach delivered by this tool. However, this required us to rethink our choice of data structures and we had move away from the functorized maps and sets in OCaml to less performant lists and sets in WhyML, in order to use the Why3 standard library for achieving automated verification. After carefully evaluating the performance between functorized data structures and less performant lists and sets, we observed that the performance differences were not large for the analyzer since the size of lists and sets were small ($n < 30$). This allowed us to choose data-structures well suited to verification over optimal asymptotic performance. We believe that this is a pragmatic choice that every software designer has to make at some point to achieve a scalable verified tool.

A drawback of the Why3 tool is its limited support of floating-point arithmetic. When we started using the Why3 floating-point support for extracting verified numerical operations, we noticed that the current version of the tool posed severe challenges in extracting floating-point literals and had limited theory for floating-point reasoning. This required us to re-think our approach for verifying properties about the physical state of the 3D printer. *We had to carefully refactor the machine configuration type for decoupling the logical controller states (whose correctness properties can be verified automatically by SMT solvers using the Why3 tool) and physical states (which involves floating-point reasoning), such that these states interact with each other minimally, allowing us to verify each of them modularly.* For the physical states, we had to define vector algebra in the Coq proof assistant and use the Flocq library [1], which provides a comprehensive floating-point theory for verifying operations on physical states. We then extracted the verified vector algebra to OCaml, for use in our analyzer. This focus on modularity in defining the states of a printer allowed us to verify some parts of the printer automatically and other parts interactively.

It is worth noting that our current design is intended to allow us to eventually verify a number of existing components in the tool that are currently hand-written. Our G-code parser makes use of a small 100-line parser combinator library instead of an external parser generator. The JSON parser also is based on a 360-line hand written stream processor that conforms to standard JSON compliance suites. Verification and implementation extraction for both parsers and their corresponding data structures can be achieved with the approach described in this paper allowing us to further reduce the scale of the hand-written trusted computing base.

5.2 Careful Modeling of Effects

A challenge of achieving a scalable verified software is the interoperability between the verified extracted parts of the software and unverified hand-written drivers. A fully verified software is challenging and there will be limits to what can be verified and what cannot be verified. While we can identify the critical components of a software and verify them, there will be parts, which involve interaction with the outside world and which pose significant challenges for verification and are therefore hand-written and form a part of the trusted computing base. These parts often have effects which propagate throughout the software and require careful modeling of these effects in the verified core components.

An approach for modeling effects in functional languages is the use of *monads*. A generic way of defining monads in proof assistants like Coq is to define a typeclass containing bind and return operations. However, this typeclass definition of monads pose a significant challenge in extraction to OCaml due to lack of support for extraction of polymorphic types from Coq to OCaml. This required us to define an instance of a monad specific to the propagation of effects in our analyzer. This ensured we were able to extract the functions with effects and inter-operate with the unverified drivers. Thus, a key lesson was to carefully evaluate the generalizability of modeling effects for specific instances, which can leverage the inherent behavior of the system to ensure smooth inter-operability between the verified and unverified components of the software.

6 Conclusion

The primary focus of this work has been the design of a verified interpreter for additive manufacturing. In this paper, we highlight some architectural lessons we learned in the process of designing this interpreter by carefully treading between verifiability and performance. We focused on decomposing the machine state into physical and non-physical states to prove correctness of each of these states using different verification approaches. This allowed us to introduce automation in verification where necessary. All the extracted data structures and linear algebra routines have already been verified. We are currently working on verifying the soundness of the interpreter step, for the functional model of the interpreter, which we extract in the development of this interpreter. Once we have the soundness theorem verified in Coq, we are planning on adding some more features into our tool like more sophisticated verification condition checks for the physical model, vendor extensions to the G-code command language, and high precision error tracking during the print process.

We would like to note that the verification process and the modular design of the interpreter was possible due to its implementation using a functional programming paradigm. It would have been challenging to peel off the states into a nice modular fashion, in an imperative programming language due to the side effects spilled all over the place.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

References

[1] 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.

[2] Adam Chlipala. 2013. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press.

[3] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 — Where Programs Meet Provers. In *Programming Languages and Systems*, Matthias Felleisen and Philippa Gardner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 125–128.

[4] Nadeem Abdul Hamid. 2024. (Nearest) Neighbors You Can Rely On: Formally Verified k-d Tree Construction and Search in Coq. In *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing* (Avila, Spain) *(SAC '24)*. Association for Computing Machinery, New York, NY, USA, 1684–1693. https://doi.org/10.1145/3605098.3635960

[5] Thomas Kramer, Frederick Proctor, and Elena Messina. 2000. The NIST RS274NGC Interpreter - Version 3. https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=823374

[6] Formal land. [n. d.]. Coq-of-ocaml. https://github.com/formal-land/coq-of-ocaml. [Accessed 22-05-2024].

[7] Mohammad Odeh, Dmitry Levin, Jim Inziello, Fluvio Lobo Fenoglietto, Moses Mathur, Joshua Hermsen, Jack Stubbs, and Beth Ripley. 2019. Methods for verification of 3D printed anatomic model accuracy using cardiac models as an example. *3D Printing In Medicine* 5 (March 2019).

[8] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2023. *Logical Foundations*. Software Foundations, Vol. 1. Electronic textbook.

Received 2024-06-03; accepted 2024-06-30