A Formal Correctness Proof for an EDF Scheduler Implementation

Florian Vanhems, Vlad Rusu, David Nowak, Gilles Grimaud

A Formal Correctness Proof for an EDF Scheduler Implementation

Florian Vanhems*, Vlad Rusu†, David Nowak*, Gilles Grimaud*
*Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
†Inria Lille-Nord Europe, France
florian.vanhems@univ-lille.fr, vlad.rusu@inria.fr, david.nowak@univ-lille.fr, gilles.grimaud@univ-lille.fr

***Abstract*—The scheduler is a critical piece of software in real-time systems. A failure in the scheduler can have serious consequences; therefore, it is important to provide strong correctness guarantees for it. In this paper we propose a formal proof methodology that we apply to an Earliest Deadline First (EDF) scheduler. It consists first in proving the correctness of the election function algorithm and then lifting this proof up to the implementation through refinements. The proofs are formalized in the Coq proof assistant, ensuring that they are free of human errors and that all cases are considered. Our methodology is general enough to be applied to other schedulers or other types of system code. To the best of our knowledge, this is the first time that an implementation of EDF applicable to arbitrary sequences of jobs has been proven correct.**

***Index Terms*—scheduler, EDF, job, proof, correction, implementation, coq, verification, formal, refinement, monad, real-time, TCB, shallow embedding**

Sources and reproductibility

We have ensured that the research described in this article is accessible and reproducible. The sources contain instructions for checking the proofs and running the scheduler on an x86_64 Linux machine. There are two back-ends: the first one runs a simulation of the scheduler compiled as a simple binary, and the second one, the actual interrupt and signal-driven scheduler running on a minimal kernel in a virtual machine. Both should be fairly easy to run, although the bare-metal scheduler requires a bit more set-up. More details about back-ends are given in Section IV-B.

The sources and instructions to reproduce our claims can be found on Github : https://github.com/2xs/pip_edf_scheduler

I. Introduction

A. Motivations

Embedded systems are present in every aspect of daily life. ATMs, production machines, self-driving vehicles, aircraft, power plants; failures in such systems can have consequences ranging from minor inconveniences to life-threatening disasters. The most critical systems should come with the strongest possible correctness guarantees. Such guarantees can be provided through formal proofs mechanized in a proof assistant.

Real-time systems are embedded systems that must perform tasks under specific time constraints. To be able to do this these systems are bundled with a scheduler that plans which task they need to perform, and when. It is essential that the scheduler makes the correct decision. This ensures, for example, that - given a reasonable task set - the time constraints are met and that situations of starvation cannot occur.

Most existing works in the literature focus on providing formal guarantees for scheduler *algorithms*. For example, [1] proves the correctness of Milner's scheduler, and [2] contains a machine–checked proof of optimality for the EDF scheduling policy. Algorithms manipulate concepts to compute a result from parameters. By contrast, their implementation on a computer uses memory and inputs/outputs provided by various hardware and software libraries. The translation of an algorithm into an executable program generally involves two steps:

- The first step is the translation of the algorithm into a programming language. Even though high-level languages facilitate this translation, it is still error-prone, due to the fact that the algorithm abstracts away many details about the entities (software or hardware) manipulated by the language. Hence, even if an algorithm is proven correct, its translation into source code may introduce errors;
- The second step is to use a compilation chain to generate executable code from the source code. Existing works such as the formally proven CompCert [3] compiler guarantee the equivalence between a C source code and the machine code generated from it, thereby ensuring that no errors are introduced in this step.

Errors in the first step can be prevented by reasoning directly on the source code. This is the approach followed in the two major projects involving formal proofs in operating systems.

The seL4 formally proven microkernel [4] contains a round-robin scheduler that refines an arbitrary scheduling policy. Recent seL4 publications on execution flow transfer and scheduling, notably [5], have focused on temporal isolation and mixed-criticality systems. This work mentions an implementation of a user-space EDF scheduler developed as a performance measurement of their main contribution, but does not present a proof of correctness. Other works [6], [7] describe a user-space implementation of an EDF scheduler running on seL4 but do not provide correctness proofs either.

CertiKOS [8] is another well-kown formally proven microkernel. A real-time extension called RT-CertiKOS contains a fixed-priority preemptive scheduler [9], with a scheduling policy specified in PROSA [10] and implemented in C and x86

assembler. A formal proof of refinement of the specification by the implementation is performed in Coq. Recently the same approach has been applied for proving an implementation of an EDF scheduler [11]. It is important to note that all schedulers in RT-CertiKOS are constrained by a *task model* that requires all tasks to be strictly periodic, and all task instances (a.k.a. *jobs*) to finish before an implicit deadline given by the beginning of the "next" corresponding period. Another salient feature of the approach followed in RT-CertiKOS is that it uses a *deep embedding*: a formalization of the semantics of a guest language (here, C and x86 assembly) into a host (here, Gallina, the language of the Coq proof assistant).

In this paper, we also present a proven implementation of an EDF scheduler in Coq. The main difference with the one existing in RT-CertiKOS is that our work is not restricted to jobs being instances of periodic tasks. Our version of EDF works for arbitrary sequences of jobs under minimal schedulability requirements. Even if jobs were instances of periodic tasks, their deadlines are not constrained to coincide with the beginning of a period. Hence, our version of EDF is strictly more expressive than the one in RT-CertiKOS, which, to our best knowledge, is currently the only other existing proof of an EDF scheduler implementation.

Another, more technical difference between our work and RT-CertiKOS is that we are using a *shallow embedding* of a subset of C into Gallina instead of a deep embedding. Shallow embedding here amounts to writing the scheduler directly in Gallina, in a syntax that is translatable, word-for-word, to C syntax. To bridge the gap between Gallina (a functional language) and C (an imperative language) we use a *state monad*, that allows one to write Gallina programs in an imperative style. A shallow embedding is more lightweight and easier to use than a deep embedding when reasoning about a program such as an EDF scheduler. Although a deep embedding is more powerful — it also allows to reason about the language, not only programs in this language — it would hinder our goal here because of the extra machinery not specifically needed to reason about a program.

B. Contributions

This paper describes the first (to our best knowledge) formally proven implementation of an EDF scheduler for arbitrary sequences of jobs. This includes jobs that are instances of periodic and sporadic tasks, with release dates having non-zero offsets with respect to periods, and deadlines that are independent of periods. The scheduler has been added on top of a formally proven microkernel [12].

C. Paper outline

Section II contains some background about scheduling. We then present in Section III our implementation of an EDF job scheduler and in particular its *election function*. Next, in Section IV we introduce the interface (or *monad*) that constitutes the environment assumed by the EDF scheduler. Section V focuses on the proof methodology, which consists in proving the correctness of the election function algorithm and lifting the correctness properties on the implementation through refinements. In Section VI, we discuss the trusted computing base (TCB), i.e., the hypotheses under which the correctness proof holds, and share ideas on how the TCB could be further reduced. We conclude in Section VII.

II. Background

In this section we review some basic facts about uniprocessor scheduling that are relevant to our approach.

The minimal objects of scheduling are *jobs*, typically characterized by a *release date* at which they become available for scheduling, a *deadline* before which their execution must be completed, and a *duration* - the amount of processor time that they require. The latter is – in general – variable and not known exactly. Instead of their exact duration, jobs are assigned a time *budget* which corresponds to their Worst-Case Execution Time (which is therefore an over-approximation of their duration).

Traditionnally, schedulers do not focus on jobs and integrate the notion of *tasks*. In such a setting, jobs are instances of *tasks* that follow specific *task models*. We will describe the most common models. In the *strictly periodic* task model, each task has a *period*. For each task, there is exactly one job per period, released at the beginning of the period and having its deadline an the end of the period. A variant of this model allows the constraint on the deadline to be relaxed, letting the job's deadline come before the end of the task's period. In the *sporadic* task model, the notion of period is changed and corresponds to the minimal time interval between two consecutive job releases. Orthogonally, there are task models that add a degree of liberty by allowing several jobs to be released during a single task period - one then talks about *reentrance*.

However, task models eventually reduce to their possible task instances, and ultimately to a concrete sequence of jobs to schedule. As such, a scheduler for arbitrary sequences of jobs is also a scheduler for a sequence of job from *any* task model. That is why we chose to implement a scheduler that disregards the notion of task and focuses directly on jobs. Specifically, we chose the EDF scheduler because it is known to be optimal [13]: if a sequence of jobs is schedulable (i.e., there exists a way to choose which job to execute at each time instant so that no job misses its deadline) then the sequence is schedulable by EDF.

III. Verified Election Function Implementation

In this section, we first give a brief but illustrated overview of our scheduler's components. We then present the verified implementation of an EDF job election function in Gallina and its translation to C.

A. Overview

In Figure 1 we give a general view of our scheduler's components. The *back-end* is the main entry point of the scheduler. It is the piece of software that handles interrupts

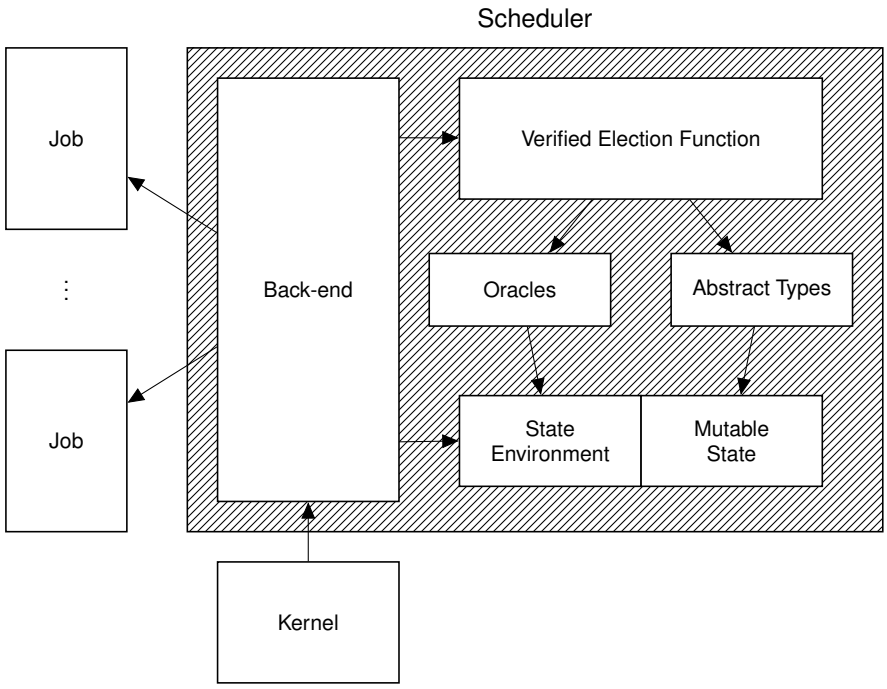

Fig. 1. Overview of the scheduler components and their interactions

coming from the kernel, calls the *election function* and executes the job chosen by the election function. One of its duties is also to update a part of the election function state that we call the *environment*. The environment contains data such as whether a job has completed its execution or has recently became available to execute. That part of the state is read-only to the election function, and can only be read through the intermediate of the *oracles*. The election function accesses and modifies the *mutable state* through its *interface*. That interface is mostly composed of data types and functions to manipulate them.

B. Election function

The election function is divided into two main parts: a first part that keeps track of the pending jobs and selects the job to be executed, and a second part that maintains the coherence of the internal state.[1]

In the first part an oracle is invoked, which returns the new jobs that needs to be scheduled. Then, the last executed job is checked, to determine whether it has finished its execution or exceeded its execution budget. If it has finished (or exceeded its execution budget a.k.a WCET - Worst-Case Execution Time) it is removed from the list of pending jobs. If the job is late, i.e., its deadline has passed, this is reported when the function returns. Then, the newly available jobs are added to the list of pending jobs, sorted by ascending deadline order, as expected from an EDF election function. Finally, the function retrieves the ID of the first job in the list (if the list is nonempty) and returns it along with the flag indicating whether the last executed job was late. A dummy value is returned if the pending jobs list is empty.

The second part of the algorithm preserves the coherence of the internal state for future calls. It starts by decreasing the time budget of the elected job by one. Then, it decreases the relative deadlines of all pending jobs by one, to account for the time spent executing the elected job. Finally, the time counter, which records how many time slots have passed since the election function was first started, is increased by one.

C. Translation

Our formally proved code is written in Gallina – the specification language of the Coq proof assistant. It is written following strict rules; ultimately, the language used is actually a shallow embedding of a subset of C in Gallina, based on a monad [14]. Basically, a monad is a way to deal with effectful computations in purely functional languages such as Gallina. It is equipped with a function `ret` that lifts a value into a computation and a function `bind` that chains computations. Those two functions must satisfy some equations stating that `bind` is an associative operation having `ret` as its neutral element. The monad is also equipped with primitive functions depending on the effects that are to be dealt with. In our case, we have primitives to deal with the environment and update the internal state of the election function.

Digger, a tool developed by the Pip kernel team [15], translates this specially crafted shallow-embedded C subset

[1]The code of the formally proved election function is located in the file `src/coq/scheduler/ElectionFunction.v`

```
Fixpoint insert_new_entries_aux timeout (new_jobs : JobSet) : RT unit :=
  match timeout with
  | 0 =>
    ret tt
  | S(timeout1) =>
    do no_more_jobs <- is_empty_list new_jobs;
    if no_more_jobs then
      ret tt
    else
      do new_job_id <- get_first_job_id new_jobs ;
      do new_entry <- create_entry_from_job_id new_job_id;
      insert_new_active_entry new_entry cmp_entry_deadline;;
      do remaining_jobs <- get_remaining_jobs new_jobs ;
      insert_new_entries_aux timeout1 remaining_jobs
  end.
```

```
void insert_new_entries_aux (unsigned int rec_bound, coq_JobSet new_jobs) {
  if (rec_bound == 0u) {
    return;
  }
  else {
    coq_CBool no_more_jobs = JobSet_is_empty_list(new_jobs);
    if (no_more_jobs) {
      return;
    }
    else {
      coq_CNat new_job_id = JobSet_get_first_job_id(new_jobs);
      coq_Entry new_entry = create_entry_from_job_id(new_job_id);
      State_insert_new_active_entry(new_entry, Entry_cmp_entry_deadline);
      coq_JobSet remaining_jobs = JobSet_get_remaining_jobs(new_jobs);
      insert_new_entries_aux(rec_bound - 1u, remaining_jobs);
      return;
    }
  }
}
```

Fig. 2. Side by side comparison of the original Gallina code (top) with its Digger translated C version (bottom)

into compilable C code. In this process, the monad (the interface with the verified code) is ironed out and implicitly integrated into our C program. During this process, the `bind` function from the monad disappears and corresponds to the notion of sequence in C. Similarly, calls to `ret` corresponds to the `return` of C. Moreover, calls to functions from the interface remain unchanged, but the translated code calls the actual function's implementation instead of their abstract model. The last feature of Digger is that it can translate specially designed recursive functions. The function must have its upper recursive bound as its first argument, and must decrement it on each call, as can be seen in Figure 2. This is an alternative to loops, which unfortunately cannot be expressed naturally in functional languages. However, the code inherits the termination guarantee from Coq, and although recursive calls are not as optimal as loops in the general case, compilers are now able to recognize them and automatically convert them to standard assembler loop structures.

Let us use the function `insert_new_entries_aux` to illustrate what the tool actually does when translating code. This function covers all of Digger's previously announced features. The Coq code and its translated counterpart are shown in Figure 2.

`Fixpoint` denotes a recursive function in Coq. The first argument of the function, `timeout`, is an upper bound on the number of items in the `new_jobs` list. `timeout` is translated into `rec_bound`. The `match` on `timeout` is translated to a `if` and `timeout1` is translated to `rec_bound - 1`. This is the most complicated transformation Digger will ever perform.

One can see that the translated code has exactly the same structure as the original code; sequences, conditional statements, and function calls remain intact. Digger is actually a word-to-word translator that does not attempt to interpret the code it processes. It operates only at the syntax level.

Even though this translation is not proved, we believe that

Digger is a sufficiently trustworthy computational base to include and rely on, and we argue that the proofs written for the original Gallina code also apply to the C code generated by Digger. However, we are aware that one can argue against such a claim. We discuss this further in Section VI.

IV. Monad model and implementation

In this section we detail how the interface of the verified code - the monad - is designed. The monad is simply a specification of how the interface should behave. We also explain the differences between the monad's model and its actual implementation. The various parts are represented in Figure 1.

We start by detailing the verified code's internal state, which is composed of two parts: a read-only part that we call the environment, and a mutable part. Then, we introduce a model-blind software that computes the information held in the environment, that we call the back-end. Next, we explain how the verified code can retrieve the information from the environment. Finally, we give details about the interface that allows the verified code to interact with the mutable part of the state.

A. Program state

The state model is divided into two parts: a part containing the values over which the algorithm has no control, its *environment*, which is immutable. These values are provided by the back-end. The second part of the model contains the values over which the algorithm has control and which are therefore mutable.

The environment is defined as a function that, given a certain timestamp, returns a list of jobs to add to the pending jobs list.

```
Definition Env : Type := CNat -> list Job.
```

The mutable state model consists of structure with two fields. The first is simply a time counter and the other is a list of *entries*. Entries are structures that associate jobs with their remaining time budget and relative deadline. The state is therefore defined as follows:

```
Record State := mk_State
    {
      now : CNat ;
      active : list Entry ;
    }.
```

The resulting state model, composed of both mutable and immutable parts, has the following type:

```
Definition RT (A : Type) : Type :=
    Env -> State -> A * State.
```

We now describe how the state is implemented. The implementation takes advantage of the fact that the election function creates a single structure for each job. Essentially, the implementation reserves additional memory space for each job to be scheduled, which the algorithm uses to store the related Entry structure.

Thus, there is an array of `coq_N` elements (where `coq_N` is the static upper bound on the number of jobs to be scheduled) that contains the initial immutable information about the pending jobs, as well as the additional mutable memory that the algorithm needs. The jobs are identified by their own index in this array. Each element of this array also contains two additional memory words. These memory bits are used to maintain the various lists required by the algorithm: the list of newly available jobs to be scheduled from the environment and maintained by the back-end, and the list of jobs pending execution from the internal state of the program, maintained by the algorithm).

So this large array has elements of type:

```
typedef struct __internal_s__ {
    struct __internal_job__   job;
    struct __internal_entry__ entry;
    int jobset_next_job_index;
    int active_next_entry_index;
} internal_t;
```

The overall state of the program consists of this big array, the heads for the two different lists, the clock counter variable from the mutable state, and a variable that indicates whether the last job completed its execution (which is manipulated by the back-end):

```
internal_t INTERNAL_ARRAY[coq_N] =
  EXAMPLE_JOB_SET;

int JOBS_ARRIVING_HEAD_INDEX = -1;
int ACTIVE_ENTRIES_HEAD_INDEX = -1;

unsigned int CLOCK = 0;
coq_CBool  JOB_DONE = false;
```

B. Back-ends

We have seen that the state is partly represented by an immutable part, that we have called its environment. This environment's purpose is to provide access to information that is not part of the algorithm per se, but which is crucial to the proper functioning of the algorithm. This includes announcing the list of newly available jobs to the algorithm, as well as announcing that a job completed its execution. These pieces of information are computed by what we call the back-end. The back-end is also responsible for calling the election function and running the elected job.

In our work, we provide two back-ends:

- The first one is a simulation of a scheduler whose main purpose is to provide insight into the inner state of the program. It calls the election function and outputs which job was chosen to execute in that time slot, as well as other relevant information about the inner state of the program. [2]
- The second one is an actual implementation of an EDF scheduler in a partition on top of the Pip kernel [15]. This implementation executes the election function and passes the execution flow to the elected job (which is in its own

[2]The source code of the simulation back-end can be found in the file: `src/partition_mockup/partition_mockup.c`.

address space). Finally, either a clock interrupt occurs (effectively interrupting the job and passing the execution flow back to the scheduler) or the job has signaled its completion to the scheduler, which waits for the next clock interrupt. This back-end uses the method outlined in [16] to transfer the execution flow.[3]

Informations computed by the back-end are made available to the verified code through oracles.

C. Oracles

There are two oracle primitives that return values managed by the back-end code and abstract to the model. The first, `jobs_arriving`, retrieves the new jobs that need to be added to the pending job list; the other, `job_terminating`, returns whether the last executed job completed its execution (if such a job exists). [4]

The monadic code can retrieve the identifiers of incoming jobs via an oracle that reads the environment. The model imposes a single constraint on the implementation: in the list of job identifiers returned by the oracle, no identifier may be greater than N, where N is a predefined static upper bound.

```
Definition jobs_arriving (N : nat)
: RT JobSet :=
  fun env s =>
    let f :=  List.filter
      (fun j =>  Nat.ltb j N)
      (map jobid (env s.(now)))
  in (f, s).
```

We would like to draw the reader's attention to the fact that the model is **purely abstract**; it does not indicate which jobs are ready for execution, since the scheduling algorithm does not rely on this information. Also note that the list of identifiers is hidden behind the abstract type `JobSet`.

In contrast, the implementation of the function that returns the oracle's prediction is quite simple:

```
static inline coq_JobSet
Primitives_jobs_arriving(coq_CNat n) {
     return JOBS_ARRIVING_HEAD_INDEX;
}
```

The implementation simply reads the index of the first arriving job from the environment. The list of incoming jobs is previously created by the back-end, so the actual list creation process depends on the back-end.

The other oracle is a function that returns whether the job with the most critical deadline (at the head of the list maintained by our scheduler) has completed its execution. The model is stricter with this oracle. It forces the implementation to return `True` if time spent executing the job is greater than its theoretical WCET.

[3]The source code of the real scheduler back-end can be found in the `src/partition` directory.

[4]Their model is available in `src/coq/model/Interface/Oracles.v`, while their C interface implementation can be found in `src/interface_implementation/include/Primitives.h`.

Here is the (nested) model definition:

```
Definition job_terminating : RT CBool :=
fun _ s => (
  (match head s.(active) with
    None => false
  | Some e =>
      let j := Jobs (e.(id)) in
        Nat.leb
          e.(cnt)
          (j.(budget) - j.(duration))
  end)
, s).
```

Note that the counter `e.(cnt)` is decremented from an initial value of `j.(budget)` each time the job is scheduled for execution.

Like the previous oracle, the implementation that returns the result of the oracle is very simple:

```
static inline coq_CBool
Primitives_job_terminating(void) {
    return JOB_DONE;
}
```

The implementation only returns the variable `JOB_DONE` from the internal state. This variable is updated by the back-end when needed.

We now review the software that allows the verified code to interact with the mutable state.

D. Abstract Types

All types used by the verified code are abstract types (black box types). These types have their own interface.

There are abstract types for Boolean values, integers, read-only job structures, and read-write job structures. Most of the primitives are simple and straightforward functions for specific data types. For example, there are 3 primitives for the boolean type `CBool` : `not`, `and`, and `or`. Here one can see the model of type CBool together with the boolean primitive `or`:

```
Definition CBool := bool.

Definition or (b1 b2 : CBool) : RT CBool :=
  ret (orb b1 b2).
```

The actual implementation is defined as follows:

```
typedef int coq_CBool;

static inline coq_CBool
CBool_or(coq_CBool b1, coq_CBool b2) {
    return b1 || b2;
};
```

Mutable structures such as type `Entry` are given primitives for creating the structure and for accessing and modifying their fields. In the same fashion, read-only data structures are only given access primitives. For example, the type `JobSet` returned by the oracle `jobs_arriving` has only three primitives : one to check for emptiness of the set, another to get the first `Job`, and a final one to get the remaining `Jobs`.

```
Definition insert_new_active_entry (entry : Entry)
                                   (comp_func : Entry -> Entry -> CBool) : RT unit :=
  fun _ s => (tt, {|
    now := now s ;
    active := (insert_Entry_aux entry (active s) comp_func);
  |}
).

Fixpoint insert_Entry_aux (entry : Entry) (entry_list : list Entry)
                          (comp_func : Entry -> Entry -> CBool) : list Entry :=
  match entry_list with
  | nil => cons entry nil
  | cons head tail =>
      match comp_func entry head with
      | true => cons entry (cons head tail)
      | false => cons head (insert_Entry_aux entry tail comp_func)
      end
  end.
```

```
void State_insert_new_active_entry (coq_Entry entry,
                                    entry_cmp_func_type entry_comp_func) {
  int *entry_index_ptr = &(ACTIVE_ENTRIES_HEAD_INDEX);
  int next_index = -1;
  while (*entry_index_ptr != -1) {
    if (entry_comp_func(entry, &(INTERNAL_ARRAY[*entry_index_ptr].entry))) {
      next_index = *entry_index_ptr;
      break;
    }
    entry_index_ptr = &(INTERNAL_ARRAY[*entry_index_ptr].active_next_entry_index);
  }
  *entry_index_ptr = entry->id;
  INTERNAL_ARRAY[entry->id].active_next_entry_index = next_index;
}
```

Fig. 3. Side by side comparison between the model of a sorted list insertion primitive and its actual implementation

The `Job` type only has primitives that return the fields of the structure. [5]

Finally, there is another set of eight primitives that interact directly with the internal state of our program. The program needs to manage a counter to keep track of time. For this, we have two primitives, a getter and a setter for the counter.

These functions are trivially modeled and implemented:

```
Definition get_time_counter
: RT CNat :=
  fun _ s => ((now s), s).

Definition set_time_counter
(counter : nat) : RT unit :=
  fun _ s => (tt,
    {|
      now    := counter ;
      active := (active s) ;
    |}
  ).
```

```
static inline coq_CNat
State_get_time_counter(){
    return CLOCK;
};

static inline void
State_set_time_counter(coq_CNat counter){
    CLOCK = counter;
};
```

The scheduler also needs to maintain a list of pending jobs. It consists of six primitives, including one primitive that determines whether jobs are present in the list, and another that returns the ID of the job located at the head of the list (if any). Two other primitives respectively add a new job to the list (in sorted form) and remove the first job from the list. The last two primitives are meant to update job structures: either

[5] All model definitions for these types are available in the `src/coq/model/Interface/Types` directory.

just the first job or the entire list. [6]

Most of these primitives are both simple to model and implement, but two of them are not so easy to implement. The first one is `insert_new_active_entry`, which inserts an entry (an internal representation of a job to execute) sorted by an arbitrary comparison function. Its model is given in the first half of Figure 3.

Given an entry and a comparison function, this model extracts the list from the current state of the program. It passes it to a (non-monadic) auxiliary recursive function that returns the list with the newly inserted entry.

This helper function then constructs a completely new list by comparing each entry from the list with the new list, using the comparison function from the parameters.

Of course, one can not afford to copy the list every time an item is inserted. The implementation uses in-place insertion in the list. The interface chosen to interact with the state prevents a mismatch between the model and reality, where one creates new lists while the other modifies the list in-place.

This code is conceptually the same as the recursive functional version, but uses loops and updates the required links between elements when a new element is inserted, rather than creating new elements and cleverly rearranging them.

The same technique was used to model and implement the function `update_active_entries`, which applies an arbitrary modification to each entry in the list. The implementation modifies the elements of the list in-place rather than creating a new list with modified copies of entries.

We argue that all the primitives described in this section are simple enough to trust that the implementation actually conforms to the model. Nevertheless, the last two primitives are arguably complex enough to shake this confidence. We discuss this point further in Section VI.

V. Proof

In this section we outline the proof of correctness of the monadic Gallina program for the EDF scheduler. We first describe our methodology, which is a top-down refinement approach between three abstraction levels. Then we show the abstraction levels and the refinements between them, focusing on why the refinement steps preserve the correctness properties proved at higher levels. Finally we give some details about the Coq implementation of the approach.

A. Methodology

We use refinement as a divide-and-conquer strategy to manage the complexity of our formal proof. We first define the EDF scheduling policy at an abstract level and show that, under appropriate assumptions, formalized below, the policy is *correct*, in the sense that any set of schedulable jobs is scheduled so all jobs complete their execution within their deadline. Then we write a functional algorithm and show that it implements the abstract scheduling policy. It follows that the functional algorithm is also correct. The final step is to write a monadic scheduler, detailed enough to be automatically translatable to C, and to show that the monadic scheduler refines the functional one; as a result, the monadic scheduler is correct as well.

1) Job model: Jobs are modelled as tuples $j = (i_j, r_j, d_j, c_j, \delta_j)$ of natural numbers, where i_j is the job's identifier, r_j is its release date (i.e., the date at which it becomes available for scheduling), d_j is the job's deadline, and c_j is the job's budget a.k.a. WCET, which is an over-approximation of the job's actual duration δ_j. The jobs satisfy the following well-formedness constraints: $r_i + c_i \leq d_i$, i.e., the deadline is large enough for the job to complete when executed alone on the processor; and $0 < \delta_j \leq c_j$, i.e., the WCET is indeed an over-approximation of the actual running time, which is itself strictly greater than zero. Moreover, it is assumed that identifiers uniquely determine jobs and that every job is released exactly once.

2) Schedulability assumption: Given any two moments t and t' such that $t < t'$, let $\Gamma_{t,t'}$ be the set of jobs to schedule in the interval $[t, t']$ (i.e. jobs j for which their release time r_j is at least t and their deadline d_j is at most t'). The schedulability condition requires that the sum of durations δ_j of the jobs to schedule in the interval $[t, t']$ is at most the length $t' - t$ of the interval.

$$\forall t, t'.\ t < t' \implies \sum_{j \in \Gamma_{t,t'}} \delta_j \leq t' - t \tag{1}$$

The fact that the schedulability of a given set of jobs implies that the set of jobs is schedulable by EDF on a single processor is a textbook result in schedulability theory ([13], pp. 33-34). It is also known that the schedulability condition has simpler forms in particular cases, e.g., for jobs that are instances of periodic tasks [13]; but in this paper we are concerned with an EDF scheduler applicable to a general set of jobs, so we shall not consider particular cases of the schedulability condition.

The study of job schedulability is a separate area of research. In this work we merely use some results from schedulability theory and prove them formally in Coq. Our proof-of-concept scheduler is bundled with a simple schedulable job set for demonstration and reproducibility purposes.

B. Abstraction levels and refinement steps

1) EDF scheduling policy: The first and most abstract level for proofs is that of the EDF scheduling policy, which can be defined as follows: for any job j and any time instant t, if the job j is running at instant t, then for any other job j' that is ready to run at the same instant, it holds that $d_j \leq d_{j'}$.

The *correctness property* for the EDF scheduling policy is stated as follows. Given a set of schedulable jobs, i.e., jobs satisfying the schedulability assumption (1), when the EDF

[6] Their model are in the `src/coq/model/Interface/Types/State.v` file, while their implementation are in the `src/interface_implementation/State.c` file. (The simplest ones are inline functions located in the corresponding header).

scheduling policy is applied, no job in the set ever exceeds its deadline.

$$\text{schedulable} \implies$$
$$\forall j, \forall t.\ \text{EdfPolicyUpTo}\ t \implies \neg\text{overdue}\ j\ t.$$

where EdfPolicyUpTo t means that the EDF policy was applied up to the instant t, and overdue j t = *true* iff the job j missed its deadline at instant t.

2) Functional EDF implements EDF policy: The first refinement step is to prove that our functional EDF election function `functional_scheduler_star` implements the EDF policy. The following property states that if the functional election function has been executed up to a certain time instant t, then the EDF policy has been applied up to that instant.

$$\forall t, \forall o, \forall s.$$
$$\texttt{functional\_scheduler\_star}\ (t) = (o, s) \implies$$
$$\text{EdfPolicyUpTo}\ (\text{now}\ s).$$

with:

s, the state of the program after executing the functional election function for t time units,
now s, extracts the current time from the state s.
o , the id of the chosen job at instant t

From this property one can derive the property expressing the correctness of the functional EDF program: given a schedulable job set and any job j from this set, that job will not exceed its deadline at the instant t if the functional EDF algorithm has been applied up to the instant t.

$$\text{schedulable} \implies$$
$$\forall t, \forall o, \forall s.$$
$$\texttt{functional\_scheduler\_star}\ (t) = (o, s) \implies$$
$$\forall i.\ \neg\text{overdue}\ i\ (\text{now}\ s).$$

3) Monadic EDF refines functional EDF: The next step consists in proving that the monadic scheduler is a refinement of the functional one. We use Hoare triples [17] for writing this property in a formal manner. A Hoare triple has the following form: $\{P\}\ c\ \{Q\}$. $\{P\}$ stands for the *preconditions*, the properties of the program state that hold before the program is executed. c represents the program, the sequence of instructions to be executed. $\{Q\}$ stands for the *postconditions*, the properties that hold after the program is executed.

The preconditions are parameterized by an observable environment *env* and by a mutable state s. The postconditions are parameterized by the return value o of the program c, and a state s', the result of executing the program c on s.

The refinement step is informally described as follows: for any given moment t, if (o, s') is the result of the execution of the monadic EDF election function `scheduler_star` on the environment E and the initial state $init$, then (o, s') is also the result of the functional EDF election function `functional_scheduler_star` at the instant t. The corresponding Hoare triple is :

$$\forall t.$$
$$\{\ \lambda\ \textit{env}\ s.\ \textit{env} = E \wedge s = \textit{init}\ \}$$
$$\texttt{scheduler\_star}\ (t)$$
$$\{\ \lambda\ o\ s'.\ \texttt{functional\_scheduler\_star}\ (t) = (o, s')\ \}$$

From this triple, one gets a Hoare triple expressing the monadic program's correctness property.

$$\textit{schedulable} \implies$$
$$\forall t.$$
$$\{\ \lambda\ \textit{env}\ s.\ \textit{env} = E \wedge s = \textit{init}\ \}$$
$$\texttt{scheduler\_star}\ (t)$$
$$\{\ \lambda\ o\ s'.\ \forall i,\ \neg\textit{overdue}\ i\ (\textit{now}\ s')\ \}$$

C. The proofs in Coq

Our Coq proofs closely follow the top-down refinement approach outlined above. First, the job model, mathematically described in Section V-A1, is encoded in Coq. The assumptions listed in V-A1 (specifically: each deadline is large enough for a job to complete its execution when running alone on the processor; job durations are greater than zero and are over-approximated by the respective budgets; identifiers uniquely determine jobs; and every job is released exactly once) are global assumptions of the proof. Arguably, the global assumptions are just reasonable well-formedness constraints.

At the highest abstraction level (EDF policy) there are additional *local* assumptions. The local assumptions are only used in the proof of correctness of the EDF scheduling policy. When the policy is subsequently refined into an algorithm the local assumptions disappear - they become Coq definitions and lemmas. Two abstract functions are locally assumed: `run`, which determines which job (if any) is running at a given moment, and `rem`, that keeps track of the remaining execution time for any given job and at any given moment. The following facts are also locally assumed about these functions, which model general facts about monoprocessor scheduling:

- at any moment in time, at most one job is running;
- up to and including its release time, the remaining execution time of a job equals its duration;
- the remaining execution time of a job decreases when the job is running, and stays constant otherwise;
- whenever there is at least a released job with non-zero remaining execution time, there is one job that is running.

Under the said local assumptions the correctness proof of the EDF policy is mostly a matter of algebraic calculations. It is relatively short: $\sim$1000 lines, mostly due to the fact that at this abstraction level one can focus on the essentials, without having to deal with implementation details.

The first refinement step, from EDF policy to EDF functional scheduler, amounts for the largest part of the proof

effort. Essentially, this is because there is a substantial abstraction gap between the EDF policy level, which is written in relatively simple algebraic terms, and the functional-scheduler level, which encodes the policy in terms of a rather detailed list data structure. The refinement consisted in defining concrete versions of the abstract functions locally assumed at the EDF policy level - *run* and *rem* - in terms of the said list structure, and in proving the properties relating these functions, which at the EDF policy level were locally assumed. Once this is done, the correctness of the EDF functional algorithm is a simple conjunction between the correctness of the EDF policy and the fact that the function scheduler implements the policy. Proving an implementation relation instead of the global correctness property on the functional scheduler is easier, since the proof obligations for the implementation relation are more detailed than the global correctness property: there are fewer details that need to be "filled in" by additional proofs.

All the properties to be proved are invariants - predicates that hold at all states reachable from an initial state, obtained by executing the functional scheduler any finite number of times. The proof required us to discover additional *inductive* invariants, which hold initially and whose truth is preserved by each execution of the scheduler. Such invariants have to be chosen so that together they imply the original invariant. This is where creativity is required from the user. Together with the functional scheduler this step amount to $\sim$1600 lines of code.

By comparison, the second (and last) refinement step is short ($\sim$500 lines). It amounts to proving that, starting from a given initial state, the reachable states generated by running the monadic EDF scheduler are included in the states generated by running the functional EDF scheduler. This was relatively easy (especially compared to the first refinement step) because the functional and monadic schedulers are relatively close in terms of abstraction level; e.g., they work with the same data structures. The difference lies in the fact that the functional scheduler computes in one large step what the monadic one computes in a sequence of many small steps. The functional algorithm is not subject to such requirements, thus, it is free to "compact" a long sequence of small steps into a large single step. By doing so the functional scheduler also saves us from the effort of proving even more additional invariants, about intermediary states that the monadic scheduler generates.

The global conclusion that the monadic EDF scheduler is correct follows from the correctness of the abstract EDF policy and the correctness of the two successive refinements.

VI. Discussion on the Trusted Computing Base

Formal proofs transfer confidence from hypotheses (or premises) to conclusions. In computer security hypotheses are called the trusted computing base. We have divided the implicit and explicit hypotheses of the trusted computing base into three categories. The first category of assumptions, the most fundamental ones, are assumptions about the correctness of the hardware and tools used to check the proof itself. The second category of assumptions includes properties expected of software libraries and runtime environments. And the last category of assumptions arises from the compilation process, more precisely from the semantics of the operations used by the source code (on which the proof is based) and the semantics of the operations used by the target code (which is actually executed). In this section, we question the trusted computing base of our work and discuss it in light of related works.

A. Fundamental barriers

In this subsection we focus on the trusted computing base that is, to the best of our knowledge, common to the whole formal code verification community.

The first common TCB is the tool used to achieve the formal verification. In our case, this is the Coq proof assistant. It is based on a logic called the calculus of inductive constructions. It assists the user in incrementally building a proof in this logic. Its critical part is its kernel – the piece of code that checks whether a tentative proof is an actual proof. One can reasonably trust the kernel of Coq and include it in the TCB.

The second, orders of magnitude larger, common TCB is the specification of the hardware running the verified code. There are two parts to this argument, that are closely related to the two steps of algorithm implementation exposed in Section I-A.

- First, most of the manufacturers do not provide a formal specification of their hardware – which implies that in order to reason on hardware specification, one has to interpret the informal specification to produce a formal one. As exposed in Section I-A, this process is tedious and error-prone. As a side-effect, it is delicate to verify software that directly rely on hardware primitives. Nonetheless, progress is made in that domain, notably with open hardware development and manufacturers beginning to provide such specifications [18].
- Second – assuming there were a formal specification of the hardware – there are no known method to prove that the hardware effectively behaves as described by its specification.

As such, and in spite of the modern hardware's increasing complexity, running code on a specific hardware implicitly includes the whole hardware in the TCB.

B. Inclusion of primitives and libraries inside the TCB

We argued in Section IV-D that the primitives we use in our election function are straightforward enough to trust *without a doubt* that their implementation indeed conforms to their specification. There are however two primitives that are arguably too big to reach such a claim : a sorted insertion primitive and a primitive that applies a modification to every element of a list.

The main reason behind our choice of interface is that the core of our contribution is the election function correctness proof. We deemed that the memory management aspect of this work was orthogonal to the properties we proved. Considering the reasonable size (respectively thirteen and seven lines of C) of the impacted primitives, we decided to treat them as trusted

libraries – like the oracles – and not to embed them further into the model.

However, in a setting where every single piece of code has to be proved (for example in a case where the proofs relies on a formal specification of the hardware), one could imagine to apply our methodology to get rid of these two primitives. The process would follow what we described in the previous sections. The first step would be to design an interface that further decomposes the two primitives, then write a monadic program that exclusively uses the broken down primitives, and finally prove that this new monadic program refines the original one.

As a side note, we want to reflect on the discussion we had in this section about our interface choices. We believe our proof methodology, and particularly the use of shallow embedding, sheds light on the assumptions brought by the Trusted Computing Base and presses us to discuss these choices. Specifically, because proof and implementation are expressed in the same language, they both share the same definition of TCB assumptions, which we believe is strongly beneficial to the proof process. In comparison, it seems straightforward to us that proofs conducted on deep embedding structures – that represents the implementation language in the proof assistant – adds a layer of obfuscation on the Trusted Computing Base.

C. Reasoning on Gallina code and compiled code trust

We believe that a toolchain that unquestionably proves every property down to the compiled code is the gold standard when working with formal methods. We strive to achieve this goal, and in this section we elaborate on the topic of trust in the compilation chain.

As mentioned in Section III-C, Digger, the automatic translation tool we use to translate the Gallina shallow embedded C to C, currently does not provide formal semantic-preservation guarantees. The lack of a such a guarantee may shed doubt on whether the formal properties proved on the Gallina code really hold for the automatically generated C code (even if we strongly believe that they do).

The next step for us is to formally prove that the semantics of the Gallina code is preserved when compiled to C. There is already related work in the community on this topic. Œuf [19] allows a large subset of idiomatic Gallina code to be compiled to C, with the guarantee that

> *valid calls to Œuf-compiled functions will behave in a manner equivalent to the user's original Gallina implementation.* – [19], Section 2.1

The Gallina program is compiled down to CompCert's Cminor, so that it can be further compiled with CompCert.

Similarly, the CertiCoq project [20] aims at verifying the compilation of *any* Gallina programs in Clight, which can then be further compiled by any C compiler, including CompCert.

An important aspect of these two tools is that they add a garbage collector to the compiled C code. A garbage collector is a complex piece of software that allows the developer to take advantage of features that are available in Gallina but not in the native C language or on trusted hardware. However, using a garbage collector usually has a negative impact on performance, and can significantly increase memory consumption. It also means that any compiled code contains a complex and unproved piece of software embedded in the trusted computing base. This trade-off is at odds with the typical requirements of embedded systems, which must be as lightweight and self-contained as possible.

Other works have tackled the problem from the other side. RefinedC [21], one of the recent efforts of the Rustbelt project, uses C code annotations to create a specification and a largely automated proof for their program. The C code itself is translated into a refined language called Caesium, which is deeply embedded in Coq and on which the proof is checked. Since their method starts from the C source code, it can be compiled directly, avoiding the performance penalty mentioned earlier. However, similarly to our own work, it is assumed that their compilation tool (front-end) from C to Coq as well as the semantics of Caesium are correct.

This type of approach has also been taken by the Verified Software Toolchain (VST) project. One of their most recent contributions is VST-Floyd [22]. VST-Floyd is an ecosystem of lemmas and tactics designed to facilitate the use of Verifiable C, a higher-order separation Hoare logic. Verifiable C is provably sound with respect to the operational semantics of CLight, which is formalized in Coq. This makes it possible to reason on almost any CLight program and provides methods and tools for end-to-end program verification.

Taking cues from all of our peers, we are currently working on a verified Gallina to CompCert C compiler. CompCert C expressions are almost identical to those of C. Our aim is to provide a compiler that will automatically translate our (restricted) Gallina shallow embedded C to CompCert C without the need to include a garbage collector. We wish to provide a way to compile CompCert C with any C compiler, including of course CompCert. We believe that this new tool will alleviate most of the proof trust concerns on our method.

VII. Conclusions and future work

Throughout this paper we have explained our methodology to prove the functional correctness of any critical code. We have concretely applied it on an EDF scheduler, in which the implementation of the election function has formally been proved correct. In particular, we have shown that our implementation of the election function is written directly in the Coq proof assistant through an embedding of C, and that we are able to translate it word-to-word to C in order to further compile it. Then, we have shown how we designed a modular interface between the verified code and its environment in order to significantly reduce the TCB of the scheduler. In Section V, we explained how we conducted our proof, by first concentrating our efforts on the election policy correctness. Then we have shown that the implementation we wrote refines the policy, inheriting its correctness property. Lastly, we discussed and provided directions on how we could further reduce the TCB.

Furthermore, we are the first to showcase, to the best of our knowledge, a formally proved correct implementation of an EDF scheduler for jobs. We also provide sources and thorough instructions that allows anyone to check our proof and execute our scheduler on their own machine.

Acknowledgments

We would like to sincerely thank Samuel Hym for reviewing a significant part of the paper. We also address our heartfelt thank you to Florence Quèbre, who provided feedback on how to improve the writing style of the document. This work was partly financed by the TinyPART project in the context of the French-German cooperation on cybersecurity (MESRI-BMBF conventions n° ANR-20-CYAL-0005 and 16KIS1395K).

References

[1] H. Korver and J. Springintveld, "A computer-checked verification of milner's scheduler," in *International Symposium on Theoretical Aspects of Computer Software*. Springer, 1994, pp. 161–178.

[2] M. Wilding, "A machine-checked proof of the optimality of a real-time scheduling policy," in *International Conference on Computer Aided Verification*. Springer, 1998, pp. 369–378.

[3] X. Leroy, S. Blazy, D. Kästner, B. Schommer, M. Pister, and C. Ferdinand, "Compcert-a formally verified optimizing compiler," in *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016.

[4] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish *et al.*, "sel4: Formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 207–220.

[5] A. Lyons, K. McLeod, H. Almatary, and G. Heiser, "Scheduling-context capabilities: A principled, light-weight operating-system mechanism for managing time," in *Proceedings of the Thirteenth EuroSys Conference*, 2018, pp. 1–16.

[6] Q. Kang, C. Yuan, X. Wei, Y. Gao, and L. Wang, "A user-level approach for arinc 653 temporal partitioning in sel4," in *2016 International Symposium on System and Software Reliability (ISSSR)*. IEEE, 2016, pp. 106–110.

[7] M. Åsberg and T. Nolte, "Towards a user-mode approach to partitioned scheduling in the sel4 microkernel," *ACM SIGBED Review*, vol. 10, no. 3, pp. 15–22, 2013.

[8] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo, "Certikos: An extensible architecture for building certified concurrent {OS} kernels," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 653–669.

[9] X. Guo, M. Lesourd, M. Liu, L. Rieg, and Z. Shao, "Integrating formal schedulability analysis into a verified os kernel," in *International Conference on Computer Aided Verification*. Springer, 2019, pp. 496–514.

[10] F. Cerqueira, F. Stutz, and B. B. Brandenburg, "Prosa: A case for readable mechanized schedulability analysis," in *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2016, pp. 273–284.

[11] X. Guo, L. Rieg, and P. Torrini, "A generic approach for the certified schedulability analysis of software systems," in *27th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2021, Houston, TX, USA, August 18-20, 2021*. IEEE, 2021, pp. 83–92. [Online]. Available: https://doi.org/10.1109/RTCSA52859.2021.00018

[12] N. Jomaa, P. Torrini, D. Nowak, G. Grimaud, and S. Hym, "Proof-oriented design of a separation kernel with minimal trusted computing base," *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.*, vol. 76, 2018. [Online]. Available: https://doi.org/10.14279/tuj.eceasst.76.1080

[13] J. A. Stankovic, M. Spuri, K. Ramamritham, and G. C. Buttazzo, *Deadline scheduling for real-time systems: EDF and related algorithms*. Springer Science & Business Media, 2012, vol. 460.

[14] P. Wadler, "Comprehending monads," in *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, ser. LFP '90. New York, NY, USA: Association for Computing Machinery, 1990, p. 61–78. [Online]. Available: https://doi.org/10.1145/91556.91592

[15] N. Jomaa, P. Torrini, D. Nowak, G. Grimaud, and S. Hym, "Proof-oriented design of a separation kernel with minimal trusted computing base," in *18th International Workshop on Automated Verification of Critical Systems (AVOCS 2018)*, 2018.

[16] F. Vanhems, N. Jomaa, S. Hym, and D. Nowak, "On the proof-oriented design of a context-switching service in the pip protokernel," in *ENTROPY 2019*, 2019.

[17] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[18] A. Reid, "Who guards the guards? formal validation of the arm v8-m architecture specification," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–24, 2017.

[19] E. Mullen, S. Pernsteiner, J. R. Wilcox, Z. Tatlock, and D. Grossman, "Oeuf: Minimizing the coq extraction tcb," in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 172–185.

[20] A. Anand, A. Appel, G. Morrisett, Z. Paraskevopoulou, R. Pollack, O. S. Belanger, M. Sozeau, and M. Weaver, "Certicoq: A verified compiler for coq," in *The third international workshop on Coq for programming languages (CoqPL)*, 2017.

[21] M. Sammler, R. Lepigre, R. Krebbers, K. Memarian, D. Dreyer, and D. Garg, "Refinedc: Automating the foundational verification of c code with refined ownership types," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 158–174. [Online]. Available: https://doi.org/10.1145/3453483.3454036

[22] Q. Cao, L. Beringer, S. Gruetter, J. Dodds, and A. W. Appel, "Vst-floyd: A separation logic tool to verify correctness of c programs," *Journal of Automated Reasoning*, vol. 61, no. 1, pp. 367–422, 2018.