Formal Verification of kLIBC with the WP Frama-C plug-in

Nuno Carvalho[1], Cristiano da Silva Sousa[1], Jorge Sousa Pinto[1], and Aaron Tomb[2]

HASLab/INESC TEC & Universidade do Minho, Portugal[1]
Galois, Inc., Portland, Oregon, USA[2]

Abstract. This paper presents our results in the formal verification of kLIBC, a minimalistic C library, using the Frama-C/WP tool. We report how we were able to completely verify a significant number of functions from `<string.h>` and `<stdio.h>`. We discuss difficulties encountered and describe in detail a problem in the implementation of common `<string.h>` functions, for which we suggest alternative implementations. Our work shows that it is presently already viable to verify low-level C code, with heavy usage of pointers. Although the properties proved tend to be shallower as the code becomes of a lower-level nature, it is our view that this is an important direction towards real-world software verification, which cannot be attained by focusing on deep properties of cleaner code, written specifically to be verified.

1 Introduction

The state-of-the-art in program verification tools based on deduction has seen great advances in recent years. This has been motivated in part by the popularity of the Design-by-Contract [1] principles, according to which program units are annotated with behavior specifications called *contracts*, that provide appropriate interfaces for *compositional* verification. On the other hand, developments in Satisfiability Modulo Theories (SMT) solvers have complemented these advances with sophisticated tools for automated theorem proving, which have made possible the automatic verification of intricate algorithms that previously required very demanding interactive proofs.

The Frama-C deductive verification plug-in WP is a tool for compositional verification of C code based on contracts. It starts with C programs annotated with behavior specifications written in a language called ACSL, and then generates a collection of *verification conditions* (VCs): proof obligations that must be valid in order for each program unit to meet its specification. A variety of back-end provers can then be used to attempt to discharge these VCs. If all VCs are shown to be valid, then the program is correct (given that the contract specified correctly covers the functional properties of the program).

An ACSL-annotated program is shown in the straightforward example of Listing 1: the `swap` C function is annotated with a precondition requiring the two pointers to be *valid* (in the sense that it can be safely accessed), which is

```
/*@
   requires \valid(a) && \valid(b);
   ensures A: *a == \old(*b);
   ensures B: *b == \old(*a);
   assigns *a,*b;
@*/
void swap(int *a,int *b){
  int tmp = *a;
  *a = *b;
  *b = tmp;
  return;
}
```

Listing 1. Swap basic example

necessary for the safe execution of the operations involving dereferencing. The postconditions on the other hand ensure the functional behavior expected of swap, and the *frame condition* states which elements of the global state are assigned during its execution.

With the development of tools such as Frama-C it is to be expected that where one would previously resort to extensive testing of code, one will now increasingly use tools to *statically verify* it. Initial applications have focused on algorithmically complex examples that are rich in 'deep' properties, but there is a clear absence in the literature of work on the verification of real-world code (initial steps in this direction with WP are reported in [2]). One class of code that could largely benefit from static verification is the code in the standard libraries of various programming languages, since more and more people depend on many widely used applications based on them.

In this paper we present our results in the formal verification of kLIBC, a minimalistic C library, using the WP plug-in of Frama-C. With this kind of verification we are treading new ground: the tools are very recent and under continued development. As such, our results should be seen as a snapshot of the state-of-the art in program verification and its applicability to real-world code.

Organization of the paper. Section 2 describes the verification and proof tools (including the underlying memory models) used in our experiments, as well as the subset of kLIBC considered. Sections 3 and 4 are the core sections of the paper, where the results obtained in the verification of functions from `<string.h>` and `<stdio.h>` are respectively reported. The verification was done bottom-up, starting with the leaf functions and then working our way up to the callers. The human effort consisted essentially in finding appropriate annotations, after that the VCs were automatically discharged. Where problems are detected, we also provide suggested corrections to the implementation of the relevant library functions. Section 5 summarizes our results and discusses the difficulties faced, and Section 6 concludes the paper. A full list of the functions analyzed can be found in Appendix A.

2 Experimental Environment

ACSL. ANSI/ISO C Specification Language (ACSL) [3] is a Behavioral Interface Specification Language (BISL) [4] for C programs, which adds a layer of first-order logic constructs on top of the well-known C syntax. As with other BISLs, such as JML [5], building on top of the programming language's syntax for boolean expressions makes it possible for programmers to easily start adding specification annotations to their programs.

All annotations are written as comments, using one of the notations `//@ ...` or `/*@ ... @*/`, for single- and multi-line annotations, respectively. The pre- and postconditions of functions are written as `\requires` and `\ensures` clauses, respectively. The memory locations that can be modified within a function call can be specified with an `\assigns` annotation (usually known as *frame condition*). Loop annotations include `loop invariant`, `loop variant` and `loop assigns`. The return value of a function can be accessed (in particular in the function's postcondition) with the `\result` clause.

All of the above annotations are fairly standard in BISLs used for deductive verification; ACSL also includes many other annotations that specifically target aspects of the C programming language. A particularly important one is the `\valid` predicate, which takes a memory location or region as argument. The intended meaning is that the content of that memory regions has been properly allocated and can thus be safely accessed. This predicate is crucial for verifying (statically) the absence of runtime memory safety violations.

Finally, the `\at` operator can be used for accessing the value of an expression at a given program state, identified by a label. As an example, the expression `\at(p, Pre)` denotes the value of the variable `p` in the pre-state of the current function. The special label `Pre` is predefined, and this expression can in fact be written equivalently as `\old(p)`. The operator can however be used with any C program label present in the program.

Frama-C and WP. Frama-C [6] is a platform dedicated to the static analysis of C source code. It has a collaborative and extensible approach that allows plug-ins to interact with each other. The most common use of Frama-C is probably as a bug-finding tool that alerts the user about dangerous code and highlights locations where errors could occur during runtime. The kind of bug-finding implemented by Frama-C aims at being correct: if there is a location in the code where an error could be generated, it should be properly reported. Users provide functional specifications written in ACSL, and Frama-C then aids them in proving that the source code is in accordance with these specifications.

Frama-C plug-ins include among others tools for calculating common source code metrics; value analysis based on abstract interpretation; program slicing; and of course deductive verification. In fact Frama-C has *two* plug-ins for deductive verification: Jessie [7] and WP [8,9]. Both plug-ins function in the same way: they convert the annotated code to a set of VCs, which are then submitted to a choice of external tools, comprising both automatic and interactive theorem provers.

```
int *p = ... ;
char *q1 = (char *)p;
char *q2 = (char *)p;
if(q1 == q2){ ... }    // CORRECT
if(*q1 == *q2){ ... }  // CORRECT
q1[2] = 0xFF;          // STILL CORRECT BUT ...
if(*p == ...)          // INCORRECT, because q1 is aliased to internal representation of p
```

Listing 2. Unsafe casts usage

In this paper we focus on the WP plug-in, since Jessie is clearly not targeted at the verification of properties of low-level code such as the code found in library functions. Moreover, WP is very actively maintained, with new versions being regularly released (for the work reported in this paper several major versions of Frama-C were used, including Oxygen and Fluorine 1, 2 and 3). We will see that a feature that has been included in the latest releases (to cope with unsafe casts) was crucial in the verification of `<stdio.h>` functions.

WP Memory models. A memory model consists of a set of data types, operations and properties that are used to construct an abstract representation of the values stored in the heap during execution of a program. The WP plug-in of Frama-C makes available to the user a number of different memory models, the simplest of which, present in every release of the plug-in, is the Hoare model, based on the weakest precondition calculus. This is a very simple model that does not support pointer operations, and is thus not suitable for our aim in this paper. The Store model was available in the Oxygen release but is not included in the more recent Fluorine releases. The heap values were stored as logical values in a global array. Support for pointer operations was fairly limited, and therefore heterogeneous type casts were not supported. Integers, floats, and pointers had to be 'boxed' into the global array and then 'unboxed' from it in order to implement read and write operations. All this boxing-unboxing was preventing automatic provers from making maximal usage of their native array theories. The Runtime model was the most powerful model included in Oxygen; it has equally been discontinued in the Fluorine release. This model was intended to be used for low-level operations, representing the heap as a wide array of bits. It was a very precise model but the price to pay for using it was high, since it generated huge VCs.

In the Fluorine releases a new model, called Typed, was introduced to replace both Store and Runtime. It makes better usage of the theories built into automated provers. The heap is represented by three memory variables, respectively holding arrays of integers, floats and addresses. This data is now indexed directly by addresses, which avoids all boxing-unboxing operations.

Very importantly, the "unsupported casts" feature of this model allows for the usage of unsafe casts, as long as they are never used to store data through a modification of the aliased memory data layout, as illustrated in Listing 2.

RTE Plug-in. This runtime error plug-in of Frama-C automatically generates annotations that can later be discharged by more powerful plug-ins such as Jessie

or WP, even though it can also be used on its own to just guard against runtime errors [10]. It is worth noting that the generated annotations may not be easily discharged, even if they can be easily generated. RTE generates annotations for:

- common runtime errors, such as division by zero, signed integer overflow or invalid memory accesses;
- unsigned integer overflows, which are considered well-defined behaviors in the C language;
- function contracts at call sites (for functions with an ACSL specification).

RTE assumes that all signed integers have a two's complement representation since it is a common implementation choice. The annotations generated are dependent of the machine where Frama-C is being executed.

Theorem Provers. The VCs generated by WP can be submitted to an interactive or automatic theorem prover. Frama-C natively supports two provers: the Alt-Ergo automatic prover and the Coq proof assistant. Other provers are supported through the Why platform[1]. In the experiments reported in this paper the following provers were used:

- Alt-ergo 0.95.1
- CVC3 2.4.1 (through the Why platform)
- Z3 4.3.1 (through the Why platform)

kLIBC kLIBC is intended to be a minimalist subset of libc, to be used with the *initramfs* file system. It is mainly used during the Linux kernel startup because at that point there is no access to the standard glibc library. It is designed for small size, minimal confusion and portability. The experiments reported in this paper focus on string-related functions from `<string.h>` and on the file API present in `<stdio.h>`. The version of kLIBC used was 2.0.2, which was released on October 5, 2012.

3 Verification of `<string.h>`

The `<string.h>` header file defines several functions to manipulate C strings and arrays. It also includes various memory handling functions. Most of these functions follow the same formula: they iterate on the string, using pointer arithmetics or an integer variable, for `n` bytes or until `'\0'` is found, performing some operation on each position.

For the functions that iterate until the end of the string is found, having access to the actual length of the string at the logical level is useful. For this we define a predicate `Length_of_str_is` as shown in Listing 3. The formula `Length_of_str_is{L}(s,n)` is true in the state identified by `L`, for nonnegative `n`, when all memory positions from 0 to `n` are valid; the final character is the null

[1] `http://why3.lri.fr`

```
/*@
  predicate Length_of_str_is{L}(char *s, integer n) =
      n >= 0 && \valid(s+(0..n)) && s[n] == 0 &&
      \forall integer k ; (0 <= k < n) ==> (s[k] != 0) ;

  axiomatic Length{
    logic integer Length{L}(char *s) reads s[..];

    axiom string_length{L}:
       \forall integer n, char *s ; Length_of_str_is(s, n) ==> Length(s) == n ;
  }
@*/
```

Listing 3. Valid string predicate and length axiom

```
/*@
    requires \exists integer i; Length_of_str_is(s,i);
    assigns \nothing;
    ensures \result == Length(s);
@*/
int strlen(const char *s) {
    const char *ss = s;

    /*@
        loop invariant BASE: \base_addr(s) == \base_addr(ss);
        loop invariant RANGE: s <= ss <= s+Length(s);
        loop invariant ZERO:
            \forall integer i; 0 <= i < (ss-s) ==> s[i] != 0;
        loop assigns ss;
        loop variant Length(s) - (ss-s);
    @*/
    while (*ss) ss++;

    //@ assert END: Length_of_str_is(s,ss-s);
    return ss - s;
}
```

Listing 4. strlen implementation and annotations

terminator `'\0'` (note that this character is present even when `n` is zero); and no other character in the string is the null terminator. Observe that the formula `\exists integer i; Length_of_str_is{L}(s,i)` holds exactly when `s` is a valid string in state `L`. With this definition, we can quickly verify the function that calculates the length of a string. Note that a logical function `Length` is also introduced, as well as an axiom linking its result to the values of `n` that satisfy the `Length_of_str_is` predicate for a given string (this enforces the existence of a single such value).

strlen. kLIBC's implementation of `strlen` and its annotations are shown in Listing 4. The contract is quite straightforward. The precondition states that only valid strings are expected by this function (this precondition is present in almost all functions from `<string.h>`). The postcondition guarantees that the result of the function is equal to the length of the string.

```
int memcmp(const void *s1, const void *s2, size_t n){
    const unsigned char *c1 = s1, *c2 = s2;
    int d = 0;

    while (n--) {
        d = (int)*c1++ - (int)*c2++;
        if (d) break;
    }
    return d;
}
```

Listing 5. Original memcmp implementation

The implementation is not the triviality one could expect. Instead, pointer arithmetic is used for loop control, where `ss-s` is the number of iterations executed. Because of this we need to define the loop invariant `RANGE` in order to guarantee that the pointer `ss` never goes out of bounds. Also, WP requires that pointers that are used in a comparison have the same base pointer, which is stated by the loop invariant `BASE`. Finally, The loop invariant `ZERO` is the one that actually allows for the contract to be proved. It states that whenever the loop condition holds, all memory positions of the array previously visited by the loop must be different from the null terminator.

The loop assigns and variant clauses are straightforward. The final assertion `END` is necessary in the Fluorine release of Frama-C: without it the contract could not be proven. What this assertion states is that, after the loop, the length of string `s` is the difference between the pointers `ss` and `s`. The resulting annotated function is fully verified with both Alt-Ergo and CVC3.

memcmp. This function compares two byte strings with at least `n` bytes of length. The implementation is shown in Listing 5 (the original implementation includes in-line assembly, which is ignored here). Just verifying the run-time execution guards uncovered an underflow in the variable `n`. This error is in fact present in multiple functions from `<string.h>`; we will now explain in detail its particular occurrence in `memcmp`.

The parameter variable `n` is declared as having type `size_t`, which is a `typedef` for an `unsigned long`, meaning the value of `n` is always larger than or equal to 0. However, in the loop's final iteration, when `n` is zero, the condition is evaluated to false but the variable is still decremented, causing an underflow. Even though this underflow does not affect the execution of the function, it is still an error, not allowing the assertion generated by WP -rte to be proven.

A proposed correction of the implementation, with the appropriate annotations, is shown in listing 6. By moving the decrement operation inside the loop, we avoid the underflow in the final iteration. The resulting annotated function is fully verified with both Alt-Ergo and CVC3.

The specification of this function requires that the memory areas pointed by `s1` and `s2` must not overlap, as otherwise the behavior would be undefined. ACSL provides the `\separated` clause for this purpose, which is here included

```
/*@
    requires n >= 0;
    requires \valid(((char*)s1)+(0..n-1));
    requires \valid(((char*)s2)+(0..n-1));
    requires \separated(((char*)s1)+(0..n-1), ((char*)s2)+(0..n-1));

    assigns \nothing;
    behavior eq:
        assumes n >= 0;
        assumes \forall integer i;
            0 <= i < n ==> ((unsigned char*)s1)[i] == ((unsigned char*)s2)[i];
        ensures \result == 0;
    behavior not_eq:
        assumes n > 0;
        assumes \exists integer i;
            0 <= i < n && ((unsigned char*)s1)[i] != ((unsigned char*)s2)[i];
        ensures \result != 0;

    complete behaviors;   // at least one behavior applies
    disjoint behaviors;   // at most one behavior applies
@*/
int memcmp(const void *s1, const void *s2, size_t n)
{
    const unsigned char *c1 = s1, *c2 = s2;
    int d = 0;
    /*@
        loop invariant N_RANGE: 0 <= n <= \at(n, Pre);
        loop invariant C1_RANGE: c1 == (unsigned char*)s1+(\at(n,Pre) - n);
        loop invariant C2_RANGE: c2 == (unsigned char*)s2+(\at(n,Pre) - n);
        loop invariant COMPARE: \forall integer i;
            0 <= i < (\at(n, Pre) - n) ==> ((unsigned char*)s1)[i] == ((unsigned char*)s2)[i];
        loop invariant D_ZERO: d == 0;
        loop assigns n, d, c1, c2;
        loop variant n;
    @*/
    while (n){
        d = (int)*c1++ - (int)*c2++;
        if (d) break;
        n--; //inserted code
    }
    return d;
}
```

Listing 6. Corrected memcmp implementation and annotations

in the precondition. Furthermore both memory areas must be valid (with length `n` bytes). Observe that, depending on the contents of the two byte strings, the result may or may not be zero. By encoding this as two different ACSL behaviors we can cover both executions.

Similarly to `strlen`, the loop invariants `N_RANGE`, `C1_RANGE` and `C2_RANGE` guarantee that the pointers never go out of bounds. The difference here is that we can specifically assert the values of the pointers `c1` and `c2` by using `n`. The loop invariants `COMPARE` and `D_ZERO` are the crucial ones for our contract, specifying that all previously iterated positions contain pairwise equal values. This implicitly means that `d==0` must always hold (otherwise the strings are not equal).

4 Verification of `<stdio.h>`

The `<stdio.h>` header file provides many functions to handle I/O operations. We aim here at verifying file functions such as `fopen`, `fclose`, and `fgetc`, i.e.

```
struct _IO_file {
    int _IO_fileno;    /* Underlying file descriptor */
    _Bool _IO_eof;     /* End of file flag */
    _Bool _IO_error;   /* Error flag */
};
typedef struct _IO_file FILE;

/*@
    predicate valid_FILE(FILE *f) = \valid(f) && f->_IO_fileno >= 0;
@*/
```

Listing 7. FILE structure definition

```
struct _IO_file_pvt {
    struct _IO_file pub; /* Data exported to inlines */
    struct _IO_file_pvt *prev, *next;
    char *buf;   /* Buffer */
    char *data;  /* Location of input data in buffer */
    unsigned int ibytes; /* Input data bytes in buffer */
    unsigned int obytes; /* Output data bytes in buffer */
    unsigned int bufsiz; /* Total size of buffer */
    enum _IO_bufmode bufmode; /* Type of buffering */
};

#define offsetof(t,m) ((size_t)&((t *)0)->m)
#define container_of(p, c, m) ((c *)((char *)(p) - offsetof(c,m)))
#define stdio_pvt(x) container_of(x, struct _IO_file_pvt, pub)
```

Listing 8. Encapsulating FILE structure and stdio_pvt macro

the *file* API. Almost all functions in this API resort to system calls, which act like black boxes, and it is thus difficult or impossible to specify what the output will be on a given input. Due to this fact, contracts tend to be quite weak. Note that even though the properties that can be verified are shallower than those considered in the previous section, they can still be extremely important – in particular, we have been able to prove various *memory safety* properties.

Since we will be working with the file API, it makes sense to start by defining a predicate that establishes the validity of a `FILE` structure. kLIBC's definition of this structure and the corresponding validity predicate are shown in Listing 7. A `FILE` structure is considered valid when both the area pointed by the pointer and the file descriptor are valid.

In reality a slightly more complex encapsulating `FILE` structure is used, see Listing 8. The `FILE` structure is kept in the field `pub` of the `_IO_file_pvt` structure. The set of all `_IO_file_pvt` structs is organized as a circular linked list. The `buf` pointer points to an area of fixed size, and `data` points to somewhere in this area, representing the current input data location in the buffer. The function `fdopen` allocates the memory necessary for this structure: memory for the structure itself, the buffer, and some extra bytes for the input buffer, as in `f = zalloc(bufoffs + BUFSIZ + _IO_UNGET_SLOP)` (kLIBC defines `BUFSIZE` as 16384 and `_IO_UNGET_SLOP` as 32).

The `valid_IO_file_pvt` predicate is defined in Listing 9. In addition to the expected safety conditions, it is stated that the values of both `ibytes` and `obytes`

```
/*@
predicate valid_IO_file_pvt(struct _IO_file_pvt *f) =
    \valid(f) && f->bufsiz == 16384 && 0 <= f->ibytes < f->bufsiz
    && 0 <= f->obytes < f->bufsiz
    && valid_FILE(&(f->pub))
    && stdio_pvt(&(f->pub)) == f
    && \separated(f, f->next, f->prev, f->buf+(0..(f->bufsiz+32-1)))
    && \valid(f->buf+(0..(f->bufsiz+32-1)))

    && f->buf <= f->data < f->buf + f->bufsiz + 32
    && \base_addr(f->data) == \base_addr(f->buf)

    && valid_IO_file_pvt_norec(f->next)
    && f->next->prev == f
    && valid_IO_file_pvt_norec(f->prev)
    && f->prev->next == f;
@*/
```

Listing 9. valid IO_file_pvt predicate

cannot exceed the actual buffer size. The `separated` clause guarantees that no memory overlapping exists between the actual file structure and its fields. This is essential to ensure that the buffer is separated from the field structure. Since they are used in a comparison operation, `data` and `buf` must have the same base address, to guarantee that the `data` pointer always points somewhere in the allocated area, as mentioned in Section 3. In order to guarantee that the circular linked list is correctly constructed, we can specify that "the next node of the previous node", and "the previous node of the next node" are both the node itself. We could use the `valid_IO_file_pvt` predicate to check the validity of the neighboring nodes. This however, would recursively check each neighboring node and WP does not support recursive predicates. Instead, we define an auxiliary predicate, similar to `valid_IO_file_pvt`, but that *does not check its neighboring nodes*. This way, whenever we check the validity of a `_IO_file_pvt` structure, its immediate neighbors are also checked. This is sufficient because all functions that require access to the linked list, only access the direct neighbors of a given `_IO_file_pvt` structure.

Functions that receive a `FILE` structure, but need to access the encapsulating `_IO_file_pvt` structure, may obtain it by resorting to the `stdio_pvt` macro, also shown in Listing 8 (the `-pp-annot` flag instructs Frama-C to process the define macros). This macro was the source of various problems when using Frama-C releases prior to Fluorine. The cast from `FILE*` to `char*` was not supported in those versions. However, the new unsafe casts option seems to handle this very well. This was crucial for the success of our efforts, since in our verification of functions from `<stdio.h>`, the `valid_IO_file_pvt` predicate was commonly used in ACSL annotations with the `stdio_pvt` macro, as in `valid_IO_file_pvt(stdio_pvt(file))`, since almost every function in the file API receives a `FILE*` pointer as argument, instead of the encapsulating structure, which is what is actually needed.

We will now consider in detail the verification of two functions from this API.

```
/*@
    requires valid_IO_file_pvt(stdio_pvt(file));
    requires -128 <= c <= 127;

    behavior fail:
        assumes stdio_pvt(file)->obytes || stdio_pvt(file)->data <= stdio_pvt(file)->buf;
        assigns \nothing;
        ensures \result == EOF;
    behavior success:
        assumes stdio_pvt(file)->obytes == 0 && !(stdio_pvt(file)->data <=
             stdio_pvt(file)->buf);
        assigns stdio_pvt(file)->ibytes, stdio_pvt(file)->data, *(\at(stdio_pvt(file)->data,
             Pre)-1);
        ensures stdio_pvt(file)->ibytes == \at(stdio_pvt(file)->ibytes, Pre) + 1;
        ensures stdio_pvt(file)->data == \at(stdio_pvt(file)->data, Pre) -1;
        ensures *(stdio_pvt(file)->data) == c == \result;

    complete behaviors; disjoint behaviors;
@*/
int ungetc(int c, FILE *file) {
    struct _IO_file_pvt *f = stdio_pvt(file);

    if (f->obytes || f->data <= f->buf) return EOF;

    *(--f->data) = c;
    f->ibytes++;

    return c;
}
```

Listing 10. ungetc implementation and specification

ungetc. This is a very simple function: it accesses some fields of the file structure, and then assigns a character back to the buffer, properly updating the `ibytes` counter. A detailed contract can be specified, because the function does not resort to system calls. The annotated function and its implementation are shown in Listing 10. Since the output of the function depends on the outcome of the conditional clause, it is adequate to define two behaviors `fail` and `success`. This function is easily verifiable in the Fluorine release with Z3, but it requires the unsafe casts option to be activated.

__fflush. Many functions in the file API rely on the `__flush` function for actually modifying (and then flushing) a file structure. Its annotated implementation is shown in Listing 11. The instruction inserted before the while loop was necessary in order to be able to specify an interval for the variable `rv` in the loop invariant. From the point of view of verification this function is very problematic, because of its dependencies. If the input buffer contains some bytes the function `fseek` is called, which in turn invokes either the system call `lseek` or `__fflush` again. On the other hand, if the output buffer is not empty its contents are written to disk with the `write` system call. Writing a deep contract for this function is thus at this point not possible, because both cases will depend on the outcome of system calls. Nevertheless we were able to specify some functional and memory safety properties. Using Z3 we were able to discharge most (but

```
/*@
    requires valid_IO_file_pvt(f);
    assigns f->ibytes, f->pub._IO_eof, f->pub._IO_error, f->obytes, errno;
    ensures \result >= -1;
@*/
int __fflush(struct _IO_file_pvt *f){
    ssize_t rv;
    char *p;

    if (__unlikely(f->ibytes)) return fseek(&f->pub, 0, SEEK_CUR);

    p = f->buf;
    rv = -1; // inserted code
    /*@
        loop invariant 0 <= f->obytes;
        loop invariant \base_addr(p) == \base_addr(f->buf);
        loop invariant -1 <= rv <= f->obytes;
        loop invariant \base_addr(f->buf) == \base_addr(f->data) == \base_addr(p);
        loop invariant f->buf <= p <= f->buf + f->bufsiz + 32;
        loop invariant \valid(p+(0..f->obytes-1));
        loop assigns f->obytes, p, f->pub._IO_eof, f->pub._IO_error, rv;
        loop variant f->obytes;
    @*/
    while (f->obytes) {
        rv = write(f->pub._IO_fileno, p, f->obytes);
        if (rv == -1){
            if (errno == EINTR || errno == EAGAIN) continue;
            f->pub._IO_error = true;
            return EOF;
        } else if (rv == 0){
            f->pub._IO_eof = true;
            return EOF;
        }

        p += rv;
        f->obytes -= rv;
    }

    return 0;
}
```

Listing 11. __fflush implementation and contract

not all) of the VCs. This partial verification is due to reasons explained above and not to the choice of prover.

5 Evaluation of Results and Difficulties

As a general remark, we note that a standard C library contains inherently low-level code, which makes it hard to verify formally, due to the presence of system calls. Verification tools of the class employed here are only capable of verifying source-level code, and of course the implementation of the system calls is in machine language – there is no way to prove their correctness. It is possible to write a basic contract for a system call, but the verification of the calling functions will always be dependent on its assumed conformance to this contract.

The Fluorine release of the WP plug-in represents a major step as the verification of properties of low-level code is concerned. In fact, in the previous Oxygen

release of WP only a few type casts were supported, such as `unsigned char*` to `char*`. In order to avoid the problems raised by this limitation we were forced to modify the code being verified, which is hardly a recommendable approach. Also, in our attempt to approach the verification of the file API with the Oxygen release we were forced to use the complex Runtime memory model, and we were unable to verify even the simplest properties of functions from `<stdio.h>`. These difficulties were eliminated in the Fluorine release, which made possible the work reported in Section 4.

A recurring problem detected in various functions from `<string.h>` is an *underflow* error present in the while loops (this is detected by all the WP releases used in our experiments). Basically, the problem is that an `unsigned` variable is decremented when its value is zero. Because of this, the RTE assertions cannot be proved. We have produced modified versions of these functions to make sure that no decrement is performed when the variable reaches zero.

In the verification of mutually recursive functions that share pointers between them we noticed that it is necessary to include in both functions' contracts the assigns clauses corresponding to the side effects produced by the code of both functions. This actually makes sense, but perhaps it would be more productive if functions could inherit assigns clauses from called functions whenever a pointer is shared (in the same way that some 'continuous' loop invariants are automatically present, such as assertions regarding variables not assigned in the loop body).

A present limitation of the WP plug-in is the lack of support for dynamic memory allocation. Even though ACSL defines clauses to deal with dynamic allocation (such as `fresh`, `allocable`, or `freeable`) these are not yet supported in the Fluorine release. According to the developers, support will be included in the next major release of the tool.

We also noted that using some provers required a heavy consumption of resources. For instance with CVC3 many threads are created but not killed in the end. We do not know if this problem is created by the prover itself, by Frama-C, or by the Why platform. Since CVC3 requires a large amount of memory very rapidly, this bug often results in forced reboots.

6 Concluding Remarks

The main goal of our experiments was to see how far one could go with verifying a low level library with Frama-C and WP. Due to the limitations described previously we ended up with partial verifications for some of the functions. Nevertheless, we were able to completely verify 14 functions out of 34 from `<string.h>`, and 13 out of 23 from the `<stdio.h>` file API. A full list of the approached functions and the present verification status (including, in the unsuccessful cases, the number of VCs left undischarged with each prover) can be found in Appendix A. The corresponding library with all the annotations is publicly available[2].

Regarding the performance of the different automated provers employed, Alt-Ergo and CVC3 seemed to be better at handling string-related functions and

[2] `https://github.com/Beatgodes/klibc_framac_wp`

behaviors, while Z3 was much more powerful dealing with the unsupported casts required to verify the file API. Within the string-related functions, Alt-Ergo was able to discharge less VCs when compared to CVC3; however, for those that were successfully discharged the computational cost was lower, and the huge amount of memory consumed by CVC3 was avoided. Since Alt-Ergo is natively supported by Frama-C, WP is able to take advantage of its built-in theories, making it a competitive option as an SMT solver.

Regardless of the partial verification results obtained for some functions, we believe we have shown without doubt that it is now viable to verify low level C code. Whatever the gravity of the problems identified in practice may be, their detection reinforces the interest of formally verifying code even when it has been widely validated by large numbers of users, as is the case of library code.

During the experiments we have identified limitations and bugs in Frama-C/WP that were properly reported to the developers. The Frama-C team is well aware of its users' needs, as evidenced by the release of the new memory model and the optional feature to support unsafe casts. The forthcoming release including support for dynamic memory allocation will very likely be another landmark in the practical applicability of deductive program verification tools.

Acknowledgment This work is funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project **FCOMP-01-0124-FEDER-020486**.

References

1. Meyer, B.: Applying "Design by Contract". IEEE Computer **25**(10) (1992)
2. Jochen Burghardt, Andreas Carben, Jens Gerlach, Kerstin Hartig, Hans Pohl, Kim Völlinger: ACSL By Example – Towards a Verified C Standard Library. DEVICESOFT project publication. Fraunhofer FIRST Institute (December 2011)
3. Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, Virgile Prevosto: ACSL: ANSI/ISO C Specification Language. (June 2013)
4. Hatcliff, J., Leavens, G.T., Leino, K.R.M., Müller, P., Parkinson, M.: Behavioral interface specification languages. ACM Comput. Surv. **44**(3) (June 2012) 16:1–16:58
5. Leavens, G., Cheon, Y.: Design by Contract with JML (2003)
6. Loïc Correnson, Pascal Cuoq, Florent Kirchner, Virgile Prevosto, Armand Puccetti, Julien Signoles and Boris Yakobowski: Frama-C User Manual. (June 2013)
7. Marché, C.: Jessie: an Intermediate Language for Java and C Verification. In Stump, A., Xi, H., eds.: Proceedings of PLPV'07, ACM (2007)
8. Patrick Baudin, Loïc Correnson, Zaynah Dargaye: WP Plug-in Manual. (June 2013)
9. Patrick Baudin, Loïc Correnson, Philippe Hermann: WP Tutorial. (September 2012)
10. Philippe Hermann and Julien Signoles: Frama-C's annotation generator plug-in. (June 2013)

A List of functions

A.1 <string.h> Functions

Function	Alt-Ergo	CVC3	Z3	Combined provers	Unsafe casts	Dependencies	Obs
bzero	✓	✓	✓		✗	memset	
memccpy	✗(15/21)	✗(18/21)	✗(13/21)	✗(18/21)	✗		Problems with PosOfChar axiom
memchr	✗(16/18)	✗(16/18)	✗(14/18)	✗(16/18)	✗		Behavior not proved, see strchr
memcmp	✓	✓	✗(15/19)		✓		
memcpy	✗(13/14)	✓	✓		✗		
memmem	✗(37/42)	✗(37/42)	✗(36/42)	✗(37/42)	✓	memcmp	Behavior not proved
memmove	✓	✓	✓		✗		
memrchr	✗(12/14)	✗(12/14)	✗(12/14)	✓	✗		
memset	✗(13/14)	✓	✓		✗		
memswap	✗(17/19)	✓	✓		✗		
strcasecmp	Bugged, does not schedule all VCs						
strcat	Dependency strchr and strcpy not verified						
strchr	✗(14/17)	✗(14/17)	✗(13/17)	✗(15/17)	✗		Behavior not proved, see memchr
strcmp	✓	✓	✗(14/22)		✗		
strcpy	✗(16/23)	✗(16/23)	✗(15/23)	✗(16/23)	✗		
strcspn	✓	✓	✗(4/5)		✗	strxspn	
strdup	Suffers from dynamic allocation problem						
strlcat	✗(15/27)	✗(15/27)	✗(13/27)	✗(15/27)	✗		Has no post-conditions
strlcpy	Bugged, does not schedule all VCs						
strlen	✓	✓	✗(7/9)		✗		
strncasecmp	✗(17/23)	✗(17/23)	✗(17/23)	✗(17/23)	✓	toUpper	
strncat	✗(12/23)	✗(12/23)	✗(9/23)	✗(12/23)	✗	strchr	
strncmp	✗(31/35)	✗(31/35)	✗(19/35)	✗(31/35)	✗		Behaviors not proved
strncpy	✗(12/16)	✗(13/16)	✗(13/16)	✗(13/16)	✗		Has no post-conditions
strndup	Suffers from dynamic allocation problem						
strnlen	✓	✓	✗(13/15)		✗		
strpbrk	✓	✓	✗(7/14)		✗	strxspn	
strrchr	✗(17/22)	✗(17/22)	✗(14/22)	✗(17/22)	✗		Behaviors not proved
strsep	✓	✓	✗(19/20)		✗	strpbrk	
strspn	✓	✓	✗(4/5)		✗	strxspn	
strstr	Dependency memmem not proved						
strtok	Dependency strxspn not proved						
strtok_r	Dependency strxspn not proved						
strxspn	✗(32/40)	✗(33/40)	✗(31/40)	✗(34/40)	✗	memset	Proved under assumption

A.2 <stdio.h> Functions

Function	Z3	Unsafe casts	Dependencies
clearerr	✓	✗	
fclose	✓	✓	fflush
fdopen	✗(19/25)	✓	
__init_stdio	✗(5/10)	✓	fdopen
feof	✓	✗	
ferror	✓	✗	
__fflush	✗(20/23)	✓	fseek
fflush	✓	✓	__fflush
fgetc	✓	✓	
fgets	n/a	✗	fgetc
fileno	✓	✗	
__parse_open_mode	✓	✗	
fopen	✓		__parse_open_mode, fdopen
fputc	Dependency _fwrite not proved		
fputs	Dependency _fwrite not proved		
_fread	✗(18/37)	✓	__fflush
fseek	✓	✓	__fflush, lseek
ftell	✗(10/11)	✓	lseek
fwrite_noflush	✗(26/39)	✓	__fflush
_fwrite	✗(29/34)	✓	
lseek	✓	✗	__llseek
rewind	✓	✓	fseek
ungetc	✓	✓	