Formal Verification of klibc with Frama-C and WP

Cristiano Rafael da Silva Sousa
Department of Informatics
University of Minho
pg22840@alunos.uminho.pt

Nuno Tiago Ferreira de Carvalho
Department of Informatics
University of Minho
pg22815@alunos.uminho.pt

In collaboration with Galois Inc.

Abstract

In this report we present the formal verification done on klibc, a minimalistic C library. With the growing need to verify programs various tools emerge. Using Frama-C with the WP plug-in we were able to completely verify a significant amount of functions from the library, showing that it is viable to verify low level C code.

1 Introduction

With the growing usage of various applications and libraries, more and more people depend on them. Where one would resort to extensive testing now we can use proper tools to statically verify code.

There is only the need to write annotations that guarantee the desired properties. The following straightforward example demonstrates this:

```
/*@ 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*

In this project we address a lower level C code from a standard library. With this kind of verification we are treading new ground: the tools are very young and under continued development.

In section 2 we explain the tools used and the underlying semantics given by the language.

Some of the approached functions are explained in detail in section 3. A full list of analyzed functions can be found in annex C.

Finally in section 4 we discuss some difficulties we encountered and we draw final conclusions in section 5

2 Experimental Environment

2.1 ACSL

ANSI/ISO C Specification Language (ACSL)[1] is a straightforward, behavioral specification language for C programs, using already known C syntax mixed with First Order Logic quantifiers ($\forall$, $\exists$).

All rules are encapsulated between `/*@ ... @*/` or with `//@ ...`. The pre- and post-conditions are written with `\requires` and `\ensures` clauses, respectively. The memory locations that are modified within a function call can be specified with an `\assigns`. The loop specification is written with `loop invariants`, `loop variant` and `loop assigns`. Note that in loops there is only one variant and one assigns.

The return value from a function can be accessed with the `\result` clause. The `\at` clause is for accessing values at a given label eg: `\at(p, Pre)` is equivalent to `\old(p)`.

It is possible to verify run time memory safety verification with the `\valid` clause.

2.2 Frama-C and WP

Frama-C is a platform dedicated to static analysis of C source code. It has a collaborative and extensible approach that allows plug-ins to be written in a way that they can interact with others.

It provides tools to measure some usual source code metrics but it is not its main function. Instead, Frama-C resembles more with bug-finding tools that alert the user of dangerous code and locations where errors could occur during runtime.

[1] `http://frama-c.com/`

In the end, what differs Frama-C from these tools is that Frama-C aims at being correct: if there's a location in the code that could generate an error, it is properly reported. The user defines functional specifications and then Frama-C aids him in proving that the source code satisfies these specifications.

Frama-C has two main plug-ins for deductive verification: Jessie and WP. Jessie was the first to be released, but it does not seem to be actively developed anymore, unlike the most recent plug-in WP which is being actively developed. This was one the reasons why we choose WP over Jessie

Both the deductive plug-ins function in the same way: they convert the code annotations to proof obligations (PO) which are then submitted to external theorem provers, both automatic and interactive.

During the course of the project several major versions of Frama-C were released: Oxygen, Fluorine 1, 2 and 3.

2.3 WP Memory models

A memory model consists in a set of data types, operations and properties that are used to abstract the values stored in the heap during the program execution.

The simplest memory model is the Hoaremodel. It is based on the Weakest Precondition Calculus and since it is very simple, it does not support pointer operations. Each variable is assigned to a logical equivalent. Clearly this model is not suitable for our project. This model is present in both old and recent versions.

In the Nitrogen release, there were two models suited for our verification purpose: Store and Runtime. We will briefly explain them, but note that they are not part of the Fluorine release.

2.3.1 Store

In Store, the heap values were stored as logical values in a global array, otherwise the POs would not be discharged by the automatic provers. However, it did not handled very well with pointer operations, and therefore heterogeneous type casts were not supported. This way, integers, floats and pointers had to be boxed into the array and unboxed from it in order to implement read and write operations. With all this boxing-unboxing, Alt-Ergo was not making maximal usage of its native array theory.

2.3.2 Runtime

This was the most powerful one. 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 it generated huge POs that were very difficult to be discharged by the automatic provers.

2.3.3 Typed

With the release of Fluorine a new model, Typed, was introduced that replaced Storeand Runtime. It makes better usage of built theories of Alt-Ergo and features of Coq.

Now the heap is represented with three memory variables, holding arrays of integers, floats and addresses, respectively. This data is now index by addresses and all boxing-unboxing operations are avoided.

Furthermore, this memory model allows for the usage of unsupported cast, as long as we manually verify that unsafe casts are never used to store data through a modification of aliased memory data layout:

```
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*

2.4 Automatic Provers

The POs generated by WP can be submitted to an interactive or automatic prover. Frama-C natively supports two provers: Alt-Ergo and Coq. Through the Why platform more automated provers can be added.

In this project we used the following provers:

- Alt-ergo 0.95.1
- CVC3 2.4.1 through Why 2 and Why 3
- Z3 4.3.1 through Why 2 and Why 3

2.5 klibc

KLIBC is intended to be a minimalist subset of the LIBC to be used with *initramfs*. 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 version used in this project is 2.0.2. We chose only to verify string related functions from <string.h> and the file API present in <stdio.h>

3 Verification

In the following sections we present the work done on verification of functions from <string.h> and <stdio.h>. The verification was done bottom-up, starting with the leaf functions and then working our way up to the callers.

3.1 <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: iterating on the string, using pointer arithmetic or an integer variable, for n bytes or until '\0' is found, and does some operation on each position. For the functions that iterate until the end of the string is found, knowing the actual length of the string would be useful. For this, we define the following predicate and axiom:

```
/*@
  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*

A string of length n is considered valid when:

- n >= 0 - when n is zero, the length is zero but still has one character: the null terminator
- All memory positions from 0 to n are valid
- The final character is the null terminator
- All characters except the last one cannot be null terminators

With this definition, we can quickly verify the function that calculates the length of a string.

3.1.1 strlen

klibc's implementation of strlen, and its annotations are as follows:

```
/*@
    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*

The contract is quite straightforward. The precondition shows that only valid strings are expected in this function. This precondition is present in almost all functions from <string.h>. The post condition guarantees that the result of the function is the length of the string, by usage of the axiom defined previously.

The implementation is not the triviality one is used to. Instead, pointer arithmetic is used as loop control, where ss - s is the number of iterations done. 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 used in comparison have the same base pointer, which can be proved as shown by the loop invariant BASE.

The loop invariant ZERO is the one that actually guarantees our contract. It states that whenever the loop condition holds, all memory positions of the array previous to the current iteration must be different from the null terminator.

Finally, the loop assigns and variant clauses are trivial.

The final assertion END was necessary in the *Fluorine* release of Frama-C. Without it, it seems that the contract cannot 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.

This function is fully verifiable with Alt-Ergo and CVC3.

3.1.2 memcmp

This function compares two byte strings of at least n bytes long. The original implementation is shown next[2]. Just verifying the run-time execution guards uncovered an underflow in the variable n.

```
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*

This variable is declared as size_t which is a typedef for an unsigned long, meaning the value of n is always larger or equal than 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.

This error is present in multiple functions from <string.h>

A proposed correction of the implementation, and its annotations is shown in listing 6.

By moving the decrement operation inside the loop, we avoid the underflow in the final iteration.

The specification for this function requires that the memory areas pointed by s1 and s2 must not overlap, as the behavior would be undefined. Using the \separated clause in the precondition we can guarantee this. Furthermore both of the memory areas must be valid for n bytes.

Depending on the content for the two byte strings, the result may or may not be zero. By encoding this into two different 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 use of n.

[2]The original implementation includes in-line assembly. This is ignored

The loop invariants COMPARE and D_ZERO are the one that guarantee our contract, by specifying that all values previous to the current iterations, are the same. This implicitly means that d == 0 must always hold, because if it does not, the strings are not equal.

```
/*@
   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;
   disjoint behaviors;
@*/
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*

3.2 <stdio.h>

The `<stdio.h>` header file provides many functions to handle I/O operations. For the purpose of our work, we're only aiming at file functions such as `fopen`, `fclose`, `fgetc`, ie. the file API.

Almost all functions from the file API resort to system calls. Due to this, contracts are very weak because the system calls act like a black box: we cannot see what happens inside. Consequently, we cannot specify what the output will be on a given input. We can however, prove various memory safety properties and loop invariants.

Since we will be working with the file API, it makes sense to define a predicate that establishes a valid `FILE` structure. *klibc*'s definition of this structure, and its predicate is as follows:

```
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*

A `FILE` structure is considered valid when the area pointed by the pointer is valid and the file descriptor is valid.

The actual `FILE` structure however, is a bit more complex:

```
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 */
};
```

Listing 8: *Actual FILE structure*

The `FILE` structure is part of the field `pub`. All `_IO_file_pvt` structs are part of a circular linked list. the `buf` pointer points to an area of fixed size and the `data` pointer to somewhere in this area, representing the current location of input data in the buffer.

If a function receives a `FILE` structure but needs the encapsulating `_IO_file_pvt` structure, it obtains it resorting to the `stdio_pvt` macro[3]:

```
#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 9: *stdio_pvt macro*

This macro was the source of various problems in the Frama-C release prior to *Fluorine*. The cast from `FILE*` to `char*` was not support at the time. However, the unsafe casts option seems to handle this very well.

The function `fdopen` allocates the memory necessary for this structure: memory for the structure itself, the buffer, and some extra bytes to the input buffer:

```
f = zalloc(bufoffs + BUFSIZ + _IO_UNGET_SLOP);
```

Listing 10: *Structure allocation*

Where *klibc* defines `BUFSIZE` as 16384 and `_IO_UNGET_SLOP` as 32.

In the following snippet we define the `valid_IO_file_pvt` predicate:

```
/*@
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 11: *valid IO_file_pvt predicate*

[3]The `-pp-annot` flag has to be used so that Frama-C can process the define macros.

Here, besides the safety options, it is stated that both the ibytes and obytes cannot exceed the actual buffer size. With the separated clause, it is guaranteed that there is no memory overlapping between the actual file structure and its fields. This is essential because the buffer must be separated from the field structure.

As mentioned in section 3.1.1, data and buf must have the same base address because they are used in comparison. Here we guarantee that the data pointer always points somewhere in the allocated area.

Usually, this predicate is used in ACSL annotations by using the stdio_pvt macro as valid_IO_file_pvt(stdio_pvt(file)), since almost all functions from the file API receive the FILE* pointer as argument instead of the encapsulating structure.

Finally, some functions require access the the neighboring nodes in the linked list, for this we use another predicate, identical to this one, but that does not check its neighboring nodes. We cannot use the same predicate to check the neighboring nodes because WP does not support recursive predicates. Since the linked list is circular, we can also specify that the next node of the previous node is, and the previous node of the next node is the node itself.

3.2.1 ungetc

This is a very simple function: it only accesses some fields of the file structure, and then it assigns a character back to the buffer, properly updating the ibytes counter.

The annotated function and its implementation is shown next:

```
/*@
   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 12: *ungetc implementation and specification*

Since the output of the function depends on the evaluation of the if clause, it is necessary to define the behaviors fail and success.

This function is easily verifiable in the Fluorine release with Z3. However the unsafe casts options must be activated.

In this particular function we can specify a detailed contract because it does not resort to system calls.

3.2.2 __fflush

The __fflush function is called by many functions from the file API, which in turn, relies on the __flush function. This is where the file structure is actually modified and data is flushed. Its implementation and annotation is shown in listing 13.

The inserted code before the while loop was necessary in order be able to specify an interval for the variable rv in the loop invariants specification.

From the point of view of verification, this function is very problematic because of its dependencies. If there are bytes on the input buffer, fseek is called, which also invokes the system call lseek. On the other hand, if there are bytes on the output buffer they are written to disk with the write system call.

Writing a good contract for this function is not currently possible because both of them will depend on the outcome of system calls.

```
/*@
   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 13: *__fflush implementation and contract*

4 Difficulties

When we started annotating `<string.h>` functions, the first problem encountered was with loop invariants. Verification suffers from invariants being to strict or too lax, so they have to be just right. This can be problematic at first because a lot of work has to be done on invariants that seems trivial at first. However, after a couple of loop annotations it becomes easier to specify and understand them.

In the Nitrogen release there only a few type casts were supported, such as `unsigned char*` to `char*`. In order to avoid this we had to change code. The suggested approach was to change implementation over function signature.

In various functions from `<string.h>` there was an underflow error present in the while loops. Basically, the problem was that an `unsigned` variable was being decremented when the value was zero. Because of this, the RTE assertions were not being proved. The solution was to change the code to make sure that no decrement was performed when the variable reached zero.

When we moved to the file API, Flourine had not yet been released, so we were forced to use the complex Runtime memory model. While using this model we were unable to verify the simplest properties from `<stdio.h>`. Fortunately, the new version was release shortly after we started working on `<stdio.h>`.

A standard C library is inherently low-level, which makes it hard to formally verify it due to system calls. The tools used are only capable of verifying C code, and the implementation of the system calls is in machine language. There is no way for us to prove the contracts for these system calls. All we can do is to write a basic contract, but calling functions' verification suffer from these weak contracts.

With functions that are mutually recursive and share pointers between them, we noticed that it is necessary for both functions to include all assigns clauses present in both of them. This actually makes sense but perhaps it would be more productive if functions would inherit the assigns clause from the functions it calls whenever a pointer is shared.

Dynamic memory allocation is currently not supported in WP. Even though there are some ACSL clauses to deal with dynamic allocation (`fresh`, `allocable`, `freeable`, etc) they are not yet part of the WP. However, according to the developers, everything is ready to support them, so they are expected to be in next major release.

We also noted that when using some provers, specially CVC3 a lot of resources are used ie. a lot of threads are created, but not killed in the end. We do not know if this is a problem with the prover itself, Frama-C or with the Why platform. Since CVC3 uses a lot of memory very quickly, when this bug occurs we are, most of the time, forced to reboot our system.

5 Concluding Remarks

A main objective of this project was to see how far could we go with verifying a low level library with Frama-C and WP.

During this project we also identified limitations and bugs in Frama-C that were properly reported to the developers. The Frama-C team is also aware of their users' needs, as evidenced by the release of the new memory model and the option to support unsafe casts. The expected support for dynamic memory allocation is something that we are anxiously waiting for.

In the end, we have successfully verified a significant amount of functions from both `<string.h>` and `<stdio.h>`, and many more are on the right path for complete verification. A full list of approached functions can be found in annex C.

However, due to some of the limitations described previously, it was not be possible to completely verify all of them, and we ended up with only a partial verification of those functions.

Nevertheless, we showed that it is viable to verify low level C code. Although the amount of time necessary to write the specification far exceeds the time needed to write the actual implementation.

Overall, we were able to completely verify 14 functions from `<string.h>` and 13 from `<stdio.h>`. During the verification, CVC3 seemed to handle better string related functions and behaviors while Z3 was much more powerful dealing with unsupported casts.

From this project, we also expect to release a public report based on this document and the annotations produced are already publicly available.

A Repository

The library with all the annotations is publicly available at the following repository:

```
https://github.com/Beatgodes/klibc_framac_wp
```

B Frama-C and WP instructions

In order to verify our annotations, Frama-C *Fluorine* and Why3 must be installed. It is also recommended to install other external provers such as Z3 and CVC3. These provers can be added to Why and to Frama-C by executing the `why3config` command. After that, they can be selected using Frama-C-GUI or by using the `-wp-prover` option.

To execute Frama-C the `-cpp-command` must be set. It is recommended to export an environment variable:

```
CPP = 'gcc -C -E -I/path/to/klibc/include -I.'
```

so that Frama-C can find the *klibc* header files. Instead of using an environment variable, the following option can also be used when running Frama-C. The `-pp-annot` flag must also be set.

```
-cpp-command="gcc -C -E -I/path/to/klibc/include -I."
```

C List of functions

C.1 <string.h>

Function	Alt-Ergo	CVC3	Z3	Combined	Unsafe casts	Dependencies	Obs
bzero	✓	✓	✓	n/a	✗	memset	
memccpy	✗(15/21)	✗(18/21)	✗(13/21)	✗(18/21)	✗		Suspect problems with PosOfChar axiom
memchr	✗(16/18)	✗(16/18)	✗(14/18)	✗(16/18)	✗		Behavior not proved, see strchr
memcmp	✓	✓	✗(15/19)	n/a	✓		
memcpy	✗(13/14)	✓	✓	n/a	✗		
memmem	✗(37/42)	✗(37/42)	✗(36/42)	✗(37/42)	✓	memcmp	Behavior not proved
memmove	✓	✓	✓	n/a	✗		
memrchr	✗(12/14)	✗(12/14)	✗(12/14)	✓	✗		
memset	✗(13/14)	✓	✓	n/a	✗		
memswap	✗(17/19)	✓	✓	n/a	✗		
strcasecmp	Bugged, does not schedule all POs						
strcat	Dependency strchr and strcpy not verified						
strchr	✗(14/17)	✗(14/17)	✗(13/17)	✗(15/17)	✗		Behavior not proved, see memchr
strcmp	✓	✓	✗(14/22)	n/a	✗		
strcpy	✗(16/23)	✗(16/23)	✗(15/23)	✗(16/23)	✗		
strcspn	✓	✓	✗(4/5)	n/a	✗	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 POs						
strlen	✓	✓	✗(7/9)	n/a	✗		
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)	n/a	✗		
strpbrk	✓	✓	✗(7/14)	n/a	✗	strxspn	
strrchr	✗(17/22)	✗(17/22)	✗(14/22)	✗(17/22)	✗		Behaviors not proved
strsep	✓	✓	✗(19/20)	n/a	✗	strpbrk	
strspn	✓	✓	✗(4/5)	n/a	✗	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	Contract proved under assumption

Table 1: *string.h functions*

C.2 <stdio.h>

In this table, results from both CVC3 and Alt-Ergo were omitted because they were not of much use in this phase.

Function	Z3	Unsafe casts	Dependencies	Obs
`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)	✓		
`__llseek`	System call			
`lseek`	✓	✗	`__llseek`	
`rewind`	✓	✓	`fseek`	
`ungetc`	✓	✓		

Table 2: *stdio.h functions*