C Library annotations in ACSL for Frama-C: experience report[1]

David R. Cok
Ian Blissard
Joshua Robbins

GrammaTech, Inc.

1 Software Specifications

To verify that software behaves as desired, one must have a description of the intended behavior. If the verification of the software is to be automated, then that description must be machine readable, in an unambiguous logical representation. Such a representation is typically called the specification, and expressed as software annotations.

For C programs, there are a variety of annotation systems. However most are fairly rudimentary, in that they only describe limited aspects of the program, such as that some variables are not null pointers, or that some assertion holds at a given program point. Other annotations systems are proprietary.

One system that seeks to be a language that can describe full behavioral requirements and be used for verifying, by proof, the correctness of a piece of software, is the ANSI-C Specification Language (ACSL) [https://frama-c.com/acsl.html]. This language is in turn based on other specification languages for other source programming languages, such as JML for Java and Spec# for C#; these are all in a family known as Behavioral Interface Specification Languages.

As part of a larger project studying annotation systems and tools, we made heavy use of ACSL and the tool set that supports it, Frama-C. The Frama-C toolkit implements both runtime and static checking of software written in ANSI-C, using ACSL annotations. Almost any substantial software makes heavy use of libraries; C programs, for example, use the C system libraries (libc). In order to verify the behavior of a program, one must have specifications for the library routines. Hence ACSL specifications for the commonly used C libraries are essential.

The Frama-C system provides some library annotations, but these were insufficient for our purpose. Accordingly we wrote additional specifications for library routines, accompanying those with test programs that would be checked by Frama-C and its wp plugin for static verification. The test programs included cases that should be provable and cases that used

[1] This material is based upon work supported by the National Science Foundation under Grant No. ACI-1314674. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

library routines incorrectly and should fail. This report describes our experience with writing and testing such specifications.

2 Standards and the GT Libc

2.1 Introduction

Programs in C have available to them a set of header files, comprising the interface to the C standard library, also called the libc. Almost all C programs expect this library to be available to them, using the functions provided for their own use. Frama-C is expected to be able to verify the goals generated by programs that employ standard C headers. To assist in verification, Frama-C includes its own version of libc. This libc is given ACSL annotations, improving Frama-C's ability to reason about programs that use functions from libc.

Several standards exist for defining what should be included as part of a libc. The most important standard is the ISO C standard. Also known as ANSI C, this standard describes the syntax and semantics of the C programming language, and also describes the minimum libc available to all ISO C programs. This is generally considered the bare minimum libc available to all C programs; almost all modern C compilers follow the ANSI standard.

Another popular standard is the POSIX standard. The "POSIX standard" is an informal name for the POSIX.1-2008 standard, which is a combination and extension of the old POSIX.1, POSIX.2, and XPG standards. The POSIX standard is itself an extension of the ISO standard, meaning all valid ISO C programs are also valid POSIX C programs. The POSIX standard is also known as the SUSv4 standard.

In addition to the functions from standards, there are popular nonstandard additions to libc that many programs expect to have available. For instance, the most popular C compiler, GCC, employs the GNU libc (called glibc), which is a POSIX-conformant libc that includes many nonstandard extensions. Another notable libc with extensions is the BSD libc; glibc and BSD libc share many of the same extensions, but are not fully compatible with each other.

The libc that comes with Frama-C is implicitly used by Frama-C to parse C programs, making the annotations to libc's functions available by default. However, this solution is not acceptable for all programs. This is due in part to the fact that Frama-C includes some but not all the headers any C program may expect to have. Frama-C includes ISO, POSIX, and GNU features as part of its libc, but not all of them. Programs that expect missing functions to be there will not parse as intended, so Frama-C's libc is not adequate for many programs. To allow more programs to be verified, we extended Frama-C's libc to include all of the ISO and POSIX standard, and some GNU extensions. We call this new libc the GT libc.

Our goals for GT libc are compatibility and annotation strength. We want GT libc to enable the parsing of ISO and POSIX-compliant programs, as well as programs that use some extensions, such as the ones provided by glibc. Any program that complies with these standards should be able to compile against the GT libc (barring some specific cases, described below). In addition, all functions provided by our libc should be properly annotated, stating the requirements and results of their execution. While not all the behavior of all functions have been modelled, all ISO and POSIX standard functions have at least a function contract in ACSL.

2.2 The ISO Standard

Frama-C libc includes all of the standard headers specified in the ISO C99 standard. It also provides annotations for some of the most commonly used headers, such as <string.h> and <stdlib.h>. GT libc extends the annotations given, adding new annotations to previously unannotated ISO functions.

Frama-C libc conforms to the C99 standard, except for a few specific headers. Frama-C cannot parse complex numbers, so <complex.h> does not have annotations or indeed declarations. <tgmath.h> is unimplemented because the standard states that its functions operate on complex numbers. Using the functions in <fenv.h> could alter the floating-point evaluation settings, which Frama-C cannot model, so that header is also unimplemented.

There is a newer version of the ISO C standard that GT libc does not conform to. C11 is the most recent version of the standard, which includes several new facilities. Some of these features require support from the parser (in our case, Frama-C), such as static assertions and generics. These features cannot be implemented in GT libc until Frama-C gains this functionality.

2.3 The POSIX Standard

Frama-C libc includes some of the functionality defined in the POSIX standard, but not all of it. It provides annotations for relatively few POSIX functions, compared to ISO functions. In addition, not all POSIX standard headers are present in Frama-C libc. GT libc adds all the missing POSIX requirements and annotates them.

The POSIX standard headers are given as a base requirement, something all POSIX-compliant systems must have, and a set of optional additions, which POSIX-compliant systems need not have. In addition, some POSIX features are marked as obsolescent. GT libc declares and annotates all base features, but not all of the options and not all of the obsolescent features. GT libc guarantees the existence of all X/OPEN system interface options (labelled the “XSI” option in the POSIX standard documentation), however.

GT libc conforms to the POSIX standard, except for one specific header: <pthread.h>. Because concurrent programs cannot be reasoned about in most of Frama-C’s modes of operation, this header has been left unimplemented. Including it currently causes a compiler error.

2.4 GNU Extensions and Glibc

Glibc is currently the most commonly used libc. It is ISO and POSIX conformant, but it adds numerous extensions. Because of its popularity, some C programs assume glibc’s extensions are available to it. Because the GNU Project maintains glibc, we call these extensions GNU extensions. Frama-C libc includes some GNU extensions, but does not annotate them. GT libc adds some more GNU extensions and annotates them, but not all GNU extensions are included yet.

Glibc adds new headers for program use. GT libc referenced the glibc Texinfo documentation for adding and annotating all the functions it includes. However, the documentation is not complete, unlike the written standards for POSIX and ISO compliance. Some behavior is left unintentionally unspecified, and some of glibc is entirely undocumented.

Some GNU extensions are not yet fully annotated in GT libc because of this. Moreover, undocumented functions have not yet been added to GT libc. Some sources of further possible documentation include the glibc man pages; these have not yet been looked into. To ensure complete compatibility with programs that expect glibc headers, the glibc source code may need to be analyzed directly.

Another issue with glibc compatibility is the fact that glibc is architecture-independent, but includes many headers that depend on the specific architecture. For example, the glibc documentation documents special headers specific to the PowerPC architecture. Therefore, Glibc compatibility includes implementing headers for all possible glibc target architectures.

Glibc also claims to be kernel-independent. It officially supports the Linux and Hurd kernels; popular, unofficial ports of glibc to other kernels exist, such as glibc for BSD and Cygwin. This implies that some of glibc's interface is also kernel-dependent. This means that glibc compatibility may mean adding functions for all possible kernels it supports as well.

Glibc also claims compatibility with portions of other standards and extensions as well, such as the BSD standard and the SVID. It does not document some of the functions it adds for compatibility reasons, so annotation of these functions is not complete.

2.5 Other Standards and Extensions

Because Glibc is the most frequently used libc, it has been the main consideration of GT libc's focus on compatibility. However, more extensions exist than GNU's extensions. Other major UNIX distributions also specify their own standards, such as BSD and System V.

On BSD systems, programs mainly use the BSD libc, which has its own set of headers. Glibc claims partial compatibility with BSD 4.4, but not complete compatibility. However, this version of BSD is considered outdated, and glibc is not as compatible with newer, derivative systems such as NetBSD, OpenBSD or FreeBSD.

The first commercial version of the UNIX OS, System V, includes the System V Interface Description (SVID), another libc standard. Again, glibc claims compatibility, but newer systems based on System V, such as HP-UX or Solaris, may not be fully compatible. GT libc does not include any headers specific to these systems.

Other libc implementations exist. For Windows systems, Microsoft Visual C includes the C Runtime Library (CRT), which is ISO- conformant (but not POSIX-conformant), and includes some headers not seen on any other OS. In addition, the Darwin kernel, the kernel used in Mac OS X and iOS, has its own libc, POSIX-conformant but different from most UNIX systems.

Currently, only a handful of features from other libc implementations have been added to GT libc. This includes some types, such as u_int32_t, used in BSD and System V. However, we do not claim any level of compatibility beyond the POSIX standard at this time.

2.6 Conclusion

Frama-C libc is only fully ISO-conformant. It lacks most POSIX and GNU functionality. GT libc adds POSIX conformance, adds some GNU extensions, and annotates most of them. GT libc can claim certain levels of standards compliance with common standards such as POSIX and ISO.

However, compatibility is not possible in all situations. Frama-C lacks the ability to parse some features, meaning that they are left out of GT libc. Overall, however, GT libc is much more complete than Frama-C libc, and therefore allows for more programs to be reasoned about. In the future, we hope to model more behavior via annotations and to increase compatibility with various libc implementations.

2.7 References

The following were useful sources of information on C standard libraries:

http://man7.org/linux/man-pages/man7/standards.7.html

http://pubs.opengroup.org/onlinepubs/9699919799

3 Overview of Frama-C Limitations

3.1 Introduction

Frama-C is a tool to parse ACSL annotations and C code, with the intention of using the ACSL annotations to generate proofs about the program's behavior. ACSL annotations are inserted into C code to inform Frama-C about the intended behavior of the program. ACSL contains language features useful for generating proofs about the program's use of memory, correctness of results, and avoidance of runtime errors.

We employed Frama-C's weakest-precondition calculus plugin (also known as the WP plugin) for our work. This plugin generates goals from our annotations, and then translates the goals to SMT-LIB through Why3 in order to verify the goals. It will then return whether or not it was able to verify a goal (stating it is valid) or not (stating it is unknown).

Frama-C, in its current state, has several limitations that reduce the number of programs that can successfully be reasoned about. These limitations may come about in several different ways, through either unresolved bugs, currently unimplemented features, or oversights in the design of ACSL.

Generally, the existence of limitations means that some annotations are weaker than they should be. Once these limitations are worked around or fixed, the annotations can be made to more closely model correct behavior. Limitations can also vary in their scope; some limitations are unique to the WP plugin, while others occur across all Frama-C plugins. In some cases, annotations cannot be applied to certain types of programs; these are the most serious kind of limitation.

The work of this project was performed primarily using Frama-C releases Neon and Sodium, but has been updated and tested using the most recent release at this writing, Silicon. Bugs and issues we encountered were reported to the Frama-C issue system and most were fixed in subsequent releases. We also reported ambiguities or errors in documentation, which also were largely corrected by the Frama-C team.

The following subsections record the limitations of Frama-C that we encountered in our work creating GT libc and using it in realistic demonstration programs. It may sound like a litany of complaints; it is not intended to be so. Rather, we document the issues we encountered as a

guide to further users (including ourselves) and as a measure of progress as the implementation of Frama-C becomes more complete. We thoroughly appreciate the Frama-C tool, the work that has been devoted to its development, and the promise it has for more correct software in the future.

3.2 Solving Timeout Issues

It is important that Frama-C be sufficiently fast to verify specifications. This is not a precise metric, but it is nonetheless important for usability that Frama-C be able to verify goals in a timely manner. To produce results that solve in an acceptable amount of time, some special considerations must be made.

Timeout occurs when a single goal takes too long to verify. To prevent timeout, one can simply use the -wp-timeout flag, setting the value to a higher number. However, this makes the proof take longer to verify for every goal, as most unknown goals take a very long time for WP to declare the goal as unknown. Without timeout, a single goal can take hours to result in an unknown verification, and in some cases, the prover will hang forever attempting to verify the unverifiable goal.

We have a few strategies to simplify proofs. First of all is the -wp-split option, which makes single goals smaller by splitting large goals into many sub-goals if possible. However, this increases the number of goals total, so it may not always save time. Another option is to annotate the programs with helper assertions, which verify sub-goals before verifying more complex goals. This can make complex goals much easier to solve, and therefore reduce the number of timeouts.

3.3 Differences in Casting between C and ACSL

C bugs concerning the casting of pointers were present in old versions of Frama-C. For instance, some pointer casts could cause a Frama-C kernel crash. For instance, casting a void* to a char* and indexing all possible elements (via the [..] mechanism) caused a crash in Frama-C Sodium. In addition, casting between different pointer types can unexpectedly fail due to problems with Frama-C's memory model (see "Memory Model Issues" below). The Silicon version does not have this problem.

3.4 Macro Substitution in ACSL

Macros are a frequently-used and important part of C. Frama-C supports the use of macros in ACSL, by finding all the defined macros in a file and expanding the macros found in the ACSL portions. While a useful feature, it has to be used with care, as improper use can cause errors in ACSL.

Since these macros are expanded in ACSL, everything used in the macro has to be valid ACSL as well. A macro that expands into a C function call cannot be used in annotations, even if that function has no side effects. This can result in errors if it isn't known at the time of annotation writing exactly what a macro will expand into. For example, the byteswap macros may not be available in Frama-C if the macros expand into the versions that employ the __built-in_bswap functions.

If macros were created with regard only to the C program, even if the macro is syntactically correct in both C and ACSL, the semantics may change. This may even create semantically invalid ACSL annotations. For instance, array types do not experience pointer decay in ACSL. Moreover, operations between pointers and arrays are illegal in ACSL, while allowed in C. Therefore, a macro used in ACSL may expand into an expression that unexpectedly violates type rules.

There are also bugs, both resolved and unresolved, in Frama-C, where macros are not always expanded. For example, if they occur directly after the set-creation '..' operator, macros will not be correctly expanded. Luckily, this can be circumvented by surrounding the macro in parenthesis. In earlier versions of Frama-C, annotating ACSL would lead to two preprocessing passes being made, which led to errors with C programs that did not expect this behavior.

3.5 Pointer Comparison Bugs

Comparing pointers in ACSL requires working around several bugs in Frama-C. To compare two pointers, Frama-C needs to verify that they both share the same pointer base. Frama-C is unable to do this itself, due to an unresolved bug in WP, so one has to manually assert that two pointers have the same base address before doing the comparison.

Another issue is with NULL. In older versions of Frama-C, WP did not compare pointers correctly if they were of different types. Since NULL is of type void*, Frama-C may not have compared pointers to NULL correctly. Therefore, programs that checked for NULL equality were not verifying as they should have. This bug has been resolved in recent versions of Frama-C.

3.6 String Literal Issues

String literals are not well-supported by Frama-C. They can be created and assigned, but the WP plugin cannot verify any goal involving them. They should have well-defined, constant contents, but verifying the values of the characters is impossible. It is also not possible to verify that any given pointer to a string literal is valid for dereferencing.

To circumvent this issue, one can employ character array literals instead. GCC expands these into a form not using a string literal, so Frama-C can reason about it. Of course, this may not be a usable alteration in all places; this changes the allocation of the literal from static space to stack space, for instance.

In our tests and demos, we had to alter portions of code to use char[] types instead of char* types wherever literals were involved. For the most part, this approach worked to reason about the contents of strings. However, in some cases, goals involving strings became complex, requiring manual assistance to validate (see "Solving Timeout Issues" above).

3.7 Memory Model Issues

The WP plugin has multiple memory models that it can use to generate goals that involve pointers and the heap. By default, the WP plugin uses the "Typed" memory model, which introduces a theory of the heap and addressing into the goals it generates. However, the WP plugin can generate goals incorrectly with this model under certain conditions.

The casting of certain types of pointers currently does not work correctly under the Typed memory model. This is due to how the model is represented internally. Casts from a pointer to a mathematical type (such as int*) may not work as expected when casted to a pointer of a storage type (such as void*). Doing these casts in a C program may cause goals to successfully validate erroneously. Furthermore, some operations allowed by C, such as casts from pointers to sufficiently large integer types, cause an explicit warning when invoked, as the Typed memory model cannot represent such concepts in the goals it generates.

Because C programs can employ casts from pointers, this limits the number of programs the WP plugin can reason about. To address this issue, Frama-C provides the "Typed+cast" memory model, which includes several changes to the Typed model in order to incorporate some pointer casts unsupported by the Typed model alone. However, the Typed+cast model does not completely ensure goal correctness for all pointer casts, and as such, is considered unsound. In the future, the Frama-C development team intends to implement the "Bytes" memory model, which should correctly address all memory model based issues with pointers.

3.8 Reals and Floats Are Treated the Same

A very subtle issue occurs when dealing with floating-point numbers at their extreme ranges. In ACSL, Frama-C deals with C floating-point numbers as if they were real numbers for all operations. This works well for mathematical proofs until concepts specific to machine (IEEE) floating-point numbers appear, such as NaNs. Real mathematics operations are all complete functions under ACSL, so there is no real-number representation of NaN, an important concept for floating-point numbers.

Another issue is that rounding errors present in floating-point numbers do not appear with ACSL reals, meaning that operations done in ACSL will vary slightly with the corresponding C code. ACSL provides the \exact and \round_error built-ins to allow one to fix these issues, but with the WP plugin, their use runs into the same issues with many built-ins (see "Unimplemented Math Built-Ins" below).

In addition, an unresolved bug exists with real numbers. Real constants given in ACSL cannot be outside the range of a C double, or a kernel crash will occur when the constant is encountered. This should not limit most programs, but it became an issue for GT libc at one point while trying to annotate the <math.h> header.

3.9 Unsupported Memberless Structs

In C, structs are required to have at least 1 member. This is due to the problematic implications of having a struct with 0 members, i.e. a 0-length type. However, the GCC compiler accepts these structs. Therefore, many programs have made use of this non-standard feature.

Notably, this can be used to implement typed opacity. A memberless struct can be used in the API, and then the definitions themselves can use a transparent, filled-in struct when actually dealing with the data. There are several opaque structs in libc that could use this feature. However, Frama-C experiences internal issues when encountering these structs.

Frama-C does not explicitly forbid memberless structs, but their use causes errors in WP, often causing crashes. Therefore, they cannot be used in programs being verified. This required us to

add dummy members to opaque structs. This change allows WP to parse the file correctly, but requires extra effort to add the member and ensure it is properly marked as an opaque type. In addition, at least one of our ACSL usage examples had to be changed by adding a dummy member to a memberless struct.

3.10 Unimplemented Math Built-ins

The ACSL manual defines a suite of built-in mathematical functions, for use in annotations. This allows arbitrary-precision math in annotations, to better model the output of some, primarily mathematical, functions. However, the WP plugin does not implement the behavior of these built-ins, and any use results in a Frama-C kernel crash.

One instance which makes this limitation apparent is the libc specifications provided by Frama-C, which makes extensive use of the nonexistent functions. As provided, the header file <math.h> would cause a crash whenever a math function was called. To fix this issue, we created our own annotations to replace some built-ins. Some, such as our replacements for the predicates \is_finite and \is_nan, replace the built-ins in some (but not all) ways. For instance, our \is_infinite and \is_nan are mutually exclusive, but other mathematical properties of infinite numbers are not implemented. Unlike the built-ins above, some built-ins, such as \sin or \exact, would be almost certainly impossible to express using ACSL logic, and our specifications for <math.h> are weaker because of it.

3.11 Unimplemented Long Doubles

Currently, Frama-C does not support the long double type as defined in ACSL. Frama-C does not give syntax errors upon encountering the type, but does warn that long doubles are not supported. Functions using this type cannot verify properties relating to the values of long double variables.

The long double type is used most extensively in <math.h>, but is also employed in other headers. We have added annotations to functions that employ long doubles, but these annotations currently do not successfully verify. For instance, our specification tests for the functions fabs and fabsf are currently successful, but our test for fabsl, the version of fabs that uses long doubles, is failing due to this issue.

3.12 Unimplemented Invariants

Another feature that the WP plugin of Frama-C does not support is invariants, both global invariants and type invariants. These could be employed to ensure that the values of certain types or globals are always in a correct format, allowing use in several headers. Moreover, we may need to specify that certain properties should always be true about the values of types or globals used by certain functions.

Currently, this does not cause any failures in verification, or any other exceptions, but it does limit our ability to create strong annotations. For example, in one of our ACSL usage examples, there is a global variable that should always be greater than zero. Global variants are unavailable, so instead, we had to find every location that modified this variable, and assert that the variable was greater than zero after modification. This solution required knowing

exactly where the global was modified, which may not always be possible. This renders our resultant annotations incapable of verifying all the programs invariant-using annotations could.

3.13 Unimplemented Allocation Clauses

In one of our ACSL usage examples, we annotated a program to discover a possible double-free bug. However, in annotating this program, we found that the WP plugin would be unable to prove that two blocks of memory, allocated by our function at different times, were not equal. Logically, they have to be, since they were each individually freshly allocated, but WP did not account for this in generating the corresponding goal. Looking into the issue, we found that this was caused by allocation clauses not being implemented as defined in ACSL.

Allocation clauses, which provide a mechanism to return fresh blocks of memory, are not implemented in WP. This is problematic for many headers, including <stdlib.h>, <string.h>, and <getopt.h>, which need this mechanism to successfully verify. Any function that allocates memory cannot express the necessary fact that the pointer it returns should not be equal to any other pointer in current existence. This makes the specifications for the affected functions weaker than they should be.

Frama-C's annotations for malloc and free use these clauses, and as such, are not adequately annotated as of now. This is an important issue, as the behavior of malloc is very important to model. Until the WP plugin implements this feature, functions that use allocation cannot be strongly annotated.

3.14 No C11 Support

Frama-C currently expects its input source code to be compatible with the C99 standard. This means that programs that employ features of the newer C11 standard cannot be parsed by Frama-C. For example, in one of our ACSL usage examples, the C11 macro _Static_assert was employed. Because this macro does not exist in C99, Frama-C could not parse the example. We had to modify the program by removing the offending usages of _Static_assert.

For some C11 additions, such as <atdatomic.h> or __STDC_NO_COMPLEX__, our libc could be altered to include them. However, some other additions, such as the macros _Static_assert or _Generic, must be implemented in the parser, as they cannot be expressed using C macros alone. Until Frama-C gains the ability to parse these additions, programs written with C11 features may not be parseable by Frama-C.

3.15 No Function Pointers

Function pointers are an important part of C, especially with more complex programs and their corresponding interfaces. In ACSL, the only logical operations that are legal with function pointers are equality and inequality. Nothing can be specified about the behavior of the referenced function.

This limitation of ACSL has a large impact on functions that employ function pointers. While ACSL can specify what function pointers are allowed to be passed in to the callee, and can require NULL not be passed as function pointer arguments to the callee, ACSL can specify little else. It cannot check if a function pointer is a valid address, it cannot require any specific behavior of function pointers, and it cannot account for function pointers having side-effects.

Therefore, any function that calls a function pointer may have to pessimistically claim it assigns every address, as the call cannot be limited to specific behaviors. This severely hinders programs that use this feature. Therefore, programs that pass or call function pointers are unprovable in most cases.

3.16 No Variadic Arguments

Through the va_list facilities provided by <stdargs.h>, a C function can be passed variadic arguments. There exist functions that employ this facility, but ACSL provides no mechanisms to reason about variadic arguments. Because of that fact, functions that employ this facility cannot be reasoned about in full.

With the … syntax, ACSL cannot reason about the underlying va_list. It cannot access the argument list in any fashion, meaning it cannot check for the validity or correct length of the arguments passed in. Attempting to list the va_list as a named argument directly is also an issue for Frama-C. When a va_list is passed to a function, Frama-C explicitly causes a user error indicating that an “implicit prototype cannot have variadic arguments”. Some of our ACSL usage examples employed this behavior, and had to be modified in order to successfully verify; this usually involves commenting out lines of code that involve calls to functions that are passed a va_list.

Variadic arguments can be very important to reason about. Some common functions that use this facility include functions like printf or ioctl. ACSL provides no facility to allow us to reason about these, making programs that use them harder or even impossible to verify.

3.17 No Complex Numbers

C defines three complex number types, corresponding to the three floating-point types. The ACSL manual does not discuss these types, and Frama-C currently cannot parse files which use complex types, returning with a syntax error. For this reason, <complex.h> explicitly causes a compilation error when included, indicating that Frama-C does not support complex number types. This means any program that makes use of complex numbers cannot be validated by Frama-C.

3.18 No Concurrency

The Frama-C WP plugin cannot reason about concurrent programs. WP assumes that all areas in memory are non-volatile, which is not the case with programs that employ concurrency. Because of this, including headers like <pthreads.h> is expressly forbidden by our libc. This is done to ensure only non-concurrent programs are reasoned about, because WP would validate goals incorrectly otherwise. For example, a pointer in a global variable may not be valid between two points in a function, even if it is not modified in between the two points, because another thread may access the global.

3.19 Conclusion

Frama-C still has several limitations that may impact how it is used. These include bugs, unimplemented features, and omissions from the ACSL language. Some limitations, like bugs, are currently being looked at and hopefully fixed; the Frama-C development team has received bug reports for all the bug-related limitations listed here. Some, however, may not be corrected

at any point, such as nonexistent features in ACSL. We continue to find more workarounds for the limitations listed above, and hope to make our library and demo annotations stronger in the future.

4 Idioms Used in GT Libc Annotations

4.1 Introduction

In our annotation of GT libc, we wished to be as consistent as possible. During the process of annotating headers and examples, we found a set of common idioms that we used in many locations throughout our annotations. Below is a list of some considerations we made while annotating GT libc.

4.2 Soundness over complete correctness

Specifying the exact behavior of any given function is not possible in the general case, as the behavior may be too complex or not possible to express in ACSL. However, specifying what inputs would invoke undefined behavior is possible for most functions. Therefore, we seek to create annotations that guarantee a program is free of undefined behavior at runtime (sound annotations) over annotations that model the behavior of a function (fully correct annotations).

This is done by specifying requirement annotations in the function contract. For example, if a pointer argument is passed in to a function that will dereference it, we specify that the argument must be a valid pointer to a correctly-sized block of memory. This may not specify exactly how it uses the values it obtains from the argument, but it states that it requires the ability to obtain them. Thus, a program that verifies with our requirement annotations is guaranteed not to invoke undefined behavior from that function call.

4.3 Restricting Assignments

A function that is not annotated and includes no definition has no restrictions on its behavior when called. Such a function could theoretically modify any memory locations available to it; this includes all globals and also locations reachable from the arguments passed in. By adding an annotation that specifies which areas of memory a function can write to, we allow programs to assume non-specified memory areas to hold their values after a call to the function.

GT libc adds an assignment annotation to all functions whose areas of modification are known. These are generally pessimistic, in that they state that a larger set of memory locations are written to than may always be affected. For instance, a function may not alter some globals if an error condition occurs; in that case, we may not specify that the write is omitted during said error condition. In some cases, we can alleviate this pessimistic assignment annotation by stating that the new value of a memory location is equal to the old value of the memory location, given the condition where the memory location is not assigned.

Assignment annotations may not be able to be added to some functions. For instance, functions that employ non-local exits or function pointers could always theoretically assign any memory location to which they have access. When encountering this class of function in GT libc, we do not include an assignment annotation.

4.4 Modelling Behavior

To model the outputs of a function, we employ *ensures* annotations. These allow us to specify facts that will hold true after a call to the function; we generally use them to specify what the output of a function will be, given the inputs.

A common behavior we specify is the behavior on error. The header <errno.h> contains the global variable *errno*, which is often set when a function detects an exceptional condition. For any function that might encounter a runtime error, we annotate it, stating that it may assign errno, and that if an error occurred, errno was set to one of a set of constant error codes, else errno was not modified.

Another type of behavior concerns validity of pointers. If a function returns a pointer, we need to indicate that the pointer points to a valid memory location, so future calls to functions that need the pointer to be valid do not fail their *requires* annotations. If a function returns a pointer, we specify whether or not it can be null, and then if it is not null, that it points to a valid location.

4.5 Axiomatics and Modelling

We employ ACSL axiomatics to assist in our annotations. This type of annotation creates a set of logical functions, and a set of axioms that are always true about the functions. This allows you to model more complex behavior in an opaque fashion.

For example, the function fopen, in <stdio.h>, returns a pointer to an opaque file object. You use fopen to pass the file pointer to other functions, which manipulate the file through the pointer. It is undefined behavior if you pass in an uninitialized file pointer to one of these functions, but it is not known what an initialized file pointer looks like internally, due to the opaqueness of the file pointer. To be able to require that a file pointer is initialized properly, we employ an axiomatic. An axiomatic function, called *file_is_open*, reads a file pointer and returns whether or not it has been initialized. After a call to fopen, we specify that the return result refers to a file pointer to which *file_is_open* would return true. Any function that manipulates a file pointer requires that *file_is_open* returns true for the given file pointer.

We also use axiomatics as shorthand for a long set of *requires* or *ensures* clauses. A good example of this is the valid_string axiomatic found in <string.h>. A string in C is a NUL-terminated series of characters, and a valid string has to be correctly terminated, or else undefined behavior could be invoked. There is an axiomatic that accepts or rejects an area of memory as a string. It requires that a valid string contain a NUL at some position, and that all the memory leading up to it is valid to read from. This allows us to add requirements that arguments to functions in <string.h> are valid strings.

4.6 Demarcating Unsupported Features

Some header files in GT libc include functions that make use of features that either cannot be parsed or would introduce unsoundness. In order to prevent users of GT libc from unintentionally employing headers that cannot be correctly used with Frama-C, we ensure that programs that make use of these headers do not compile. We implement this with the #error preprocessor directive, which causes a compiler error when parsed by the C preprocessor.

With this method, any program that includes a problematic header cannot be verified with Frama-C. In addition, the compiler fails with a helpful error message instead of a Frama-C kernel crash, stating the Frama-C does not support the header. This gives the user definite feedback on why the program will not compile.

4.7 Non-Standard Control Flow

Functions that halt the program, do not terminate, or do not return to the calling function for other reasons are annotated with the post-condition false. When possible, this is a conditional behavior to allow some use of the function. However, this pessimistic approach must be taken to ensure soundness. This post-condition introduces the contradiction to show that the result of calling this function may not resume execution as expected.

4.8 Conclusion

GT libc employs several common patterns in its annotations. These patterns allow us to consistently generate annotations fit for use by most programs. While many annotations in GT libc do not yet completely describe the corresponding function, we hope to continue using these idioms to strengthen our annotations.

5 Review of Specifications

5.1 Overview

To facilitate formal verification of real world C applications, annotations are provided for standard libraries, in order to specify their behavior. These specifications were originally provided by Frama-C, and then expanded by GrammaTech. The three most common standards were targeted: The International Organization for Standards (ISO), Portable Operating System Interface (POSIX), and GNU extensions for libc were evaluated. We chose to include obsolete headers for compatibility, and to omit some optional features.

The tables below summarize the contributions we made to annotating these libraries.

ISO header files

File Name	Standard Introduced	Frama-C Specifications	GT Specifications
assert.h	ISO	WEAK	WEAK
complex.h*	ISO	NONE	NONE
ctype.h	ISO	NONE	STRONG
errno.h	ISO	N/A	N/A
fenv.h*	ISO	NONE	NONE
float.h	ISO	N/A	N/A
inttypes.h	ISO	WEAK	WEAK
iso646.h	ISO	N/A	N/A
limits.h	ISO	N/A	N/A
locale.h	ISO	WEAK	WEAK
math.h	ISO	WEAK	WEAK
setjmp.h	ISO	WEAK	WEAK
signal.h	ISO	NONE	WEAK
stdarg.h	ISO	NONE	NONE
stdbool.h	ISO	N/A	N/A
stddef.h	ISO	N/A	N/A
stdint.h	ISO	N/A	N/A
stdio.h	ISO	WEAK	WEAK
stdlib.h	ISO	WEAK	WEAK
string.h	ISO	STRONG	STRONG
tgmath.h*	ISO	NONE	NONE
time.h	ISO	WEAK	WEAK
wchar.h	ISO	NONE	WEAK
wctype.h	ISO	NONE	STRONG

POSIX header files

File Name	Standard Introduced	Frama-C Specifications	GT Specifications
aio.h	POSIX	NONE	WEAK
argz.h	POSIX	NONE	WEAK
arpa/inet.h	POSIX	WEAK	WEAK
cpio.h	POSIX	N/A	N/A
dirent.h	POSIX	WEAK	WEAK
dlfcn.h	POSIX	NONE	WEAK
fcntl.h	POSIX	WEAK	WEAK
fmtmsg.h	POSIX	NONE	WEAK
fnmatch.h	POSIX	NONE	WEAK
ftw.h	POSIX	NONE	WEAK
glob.h	POSIX	NONE	WEAK
grp.h	POSIX	NONE	WEAK
langinfo.h	POSIX	NONE	WEAK
lconv.h	POSIX	NONE	WEAK
libgen.h	POSIX	NONE	WEAK
monetary.h	POSIX	NONE	WEAK
ndbm.h	POSIX	NONE	WEAK
nedtb.h	POSIX	NONE	WEAK
net/if.h	POSIX	NONE	WEAK
netinet/in.h	POSIX	N/A	N/A
nl_types.h	POSIX	NONE	WEAK
poll.h	POSIX	NONE	WEAK
pthread.h*	POSIX	NONE	NONE
pwd.h	POSIX	NONE	WEAK
regex.h	POSIX	NONE	WEAK
sched.h	POSIX	NONE	WEAK
search.h	POSIX	NONE	WEAK
semaphore.h	POSIX	NONE	WEAK
strings.h	POSIX	NONE	WEAK
stropts.h	POSIX	NONE	WEAK
sys/ipc.h	POSIX	NONE	WEAK
sys/mman.h	POSIX	NONE	WEAK
sys/msg.h	POSIX	NONE	WEAK
sys/resource.h	POSIX	WEAK	WEAK
sys/select.h	POSIX	NONE	WEAK
sys/sem.h	POSIX	NONE	WEAK
sys/shm.h	POSIX	NONE	WEAK
sys/socket.h	POSIX	WEAK	WEAK
sys/stat.h	POSIX	NONE	WEAK

sys/statvfs.h	POSIX	NONE	WEAK
sys/time.h	POSIX	WEAK	WEAK
sys/times.h	POSIX	NONE	WEAK
sys/uio.h	POSIX	WEAK	WEAK
sys/un.h	POSIX	N/A	N/A
sys/utsname.h	POSIX	NONE	WEAK
sys/wait.h	POSIX	NONE	WEAK
syslog.h	POSIX	WEAK	WEAK
tar.h	POSIX	N/A	N/A
termios.h	POSIX	NONE	WEAK
ulimit.h	POSIX	NONE	WEAK
unistd.h	POSIX	NONE	WEAK
utmpx.h	POSIX	NONE	WEAK
wordexp.h	POSIX	NONE	WEAK

GNU Header files

File Name	Standard Introduced	Frama-C Specifications	GT Specifications
argp.h	GNU	NONE	WEAK
byteswap.h	GNU	N/A	N/A
crypt.h	GNU	NONE	WEAK
elf.h	GNU	N/A	N/A
endian.h	GNU	N/A	N/A
envz.h	GNU	NONE	WEAK
err.h	GNU	NONE	STRONG
error.h	GNU	NONE	STRONG
execinfo.h*	GNU	NONE	NONE
features.h	GNU	N/A	N/A
fstab.h	GNU	NONE	WEAK
getopt.h	GNU	WEAK	WEAK
ifaddrs.h	GNU	NONE	NONE
libintl.h	GNU	NONE	WEAK
malloc.h	GNU	NONE	WEAK
mcheck.h	GNU	NONE	WEAK
mntent.h	GNU	NONE	WEAK
netinet/in_system.h	GNU	NONE	NONE
netinet/ip.h	GNU	NONE	NONE
netinet/ip_cmp.h	GNU	NONE	NONE
obstack.h	GNU	NONE	WEAK
printf.h	GNU	NONE	WEAK
pty.h	GNU	NONE	WEAK
sgtty.h	GNU	NONE	WEAK
stdio_ext.h	GNU	NONE	WEAK
sys/auxv.h	GNU	NONE	WEAK
sys/ioctl.h	GNU	NONE	WEAK
sys/mount.h	GNU	NONE	WEAK
sys/param.h	GNU	N/A	N/A
sys/platform/ppc.h	GNU	NONE	WEAK
sys/statfs.h	GNU	NONE	NONE
sys/syscall.h	GNU	N/A	N/A
sys/sysctl.h	GNU	NONE	WEAK
sys/sysinfo.h	GNU	NONE	WEAK
sys/timex.h	GNU	N/A	N/A
sys/vfs.h	GNU	N/A	N/A
sys/vlimit.h	GNU	NONE	WEAK
sys/vtimes.h	GNU	NONE	WEAK
uchar.h	GNU	NONE	NONE
ucontext.h	GNU	NONE	NONE
utmp.h	GNU	NONE	WEAK

5.2 Weak Specifications

The majority of specifications provided do not completely describe all behavior. The current weakness in specifications can be divided into three major causes. First, there are limitations in the automated solver. Both Frama-C and the WP plugin have inherent limitations, as well as unresolved bugs which limit reasoning. Second, external interactions such as file I/O cannot be precisely defined, without knowing the contents. Finally, in some cases, the specification could be improved, but is difficult to do so.

5.2.1 Unavailable Features

There are limitations in Frama-C and WP which weaken the specifications that can be proven automatically. For instance, there is no mechanism to reason about concurrency, asynchronous transfer of control, or non-local exits. Additionally, there are features of ACSL which are not implemented, restricting C features such as dynamic allocation, variadic arguments, and 128 bit types.

Unusual control flow will cause errors or unsound reasoning, and must be avoided in the Frama-C framework. Concurrency is disallowed, and the header pthread.h may not be used. Furthermore, any functions which introduce threads or shared memory will cause a (intentional) contradiction. Six of the evaluated headers include functions that asynchronously transfer control. These functions will have weak specifications. Finally, the header setjmp.h allows non-local exits, resulting in no frame condition.

ISO	signal.h					
POSIX	aio.h	sched.h	semaphore.h	sys/socket.h	sys/msg.h	sys/sem.h

Headers which rely on asynchronous transfer of control.

ACSL features which have not been implemented in WP, including C features not implemented in Frama-C, result in a necessary weakening of specifications. The built in predicates for dynamic allocation are not available, hindering reasoning about memory separation. Furthermore, variadic arguments, function pointers, and 96 or 128 bit types have limited or no functionality in Frama-C. A program using these features will be unable to produce sound validation.

ISO	stdlib.h			
GNU	malloc.h	argz.h	envz.h	obstack.h

Headers which use dynamic allocation.

ISO	stdarg.h	stdio.h	
POSIX	strops.h		
GNU	sys/ioctl.h	sys/sysctl.h	err.h

Headers which use variadic arguments.

ISO	math.h
POSIX	search.h
GNU	printf.h

Headers which require 96 or 128 bit types.

The math functions have additional weaknesses that may be resolved with further ACSL implementation in WP. ACSL defines arithmetic built-ins such as \cos which would allow a more precise specification. Instead, reasonable ranges are provided, or results for simple cases, such as *cos(0)=1*.

5.2.2 Weakness from External Interactions

Programs often interact with parts of the system that are outside of the ACSL program model. Therefore, weak specifications must be generated to account for uncertainty. The two primary sources of external interaction are though the file system and networking. Parts of the file system interaction can be modeled, such as the open/closed status of a file. However, other aspects, including the file's contents, cannot be modelled. Similarly, sockets used in networking can be modeled, but the data read is uncertain. We expect that specifications in these areas will not improve.

ISO	POSIX		GNU
ndbm.h	netdb.h	sys/resource.h	stdio_ext.h
stdio.h	nl_types.h	utmp.h	sys/ioctl.h
fcntl.h	poll.h	utmpx.h	fstab.h
	pwd.h	Ifaddrs.h	ftw.h
	sys/select.h	sys/ipc.h	fmtmsg.h
	sys/stat.h	sys/shm.h	sys/mount.h
	sys/statvfs.h	sys/uio.h	sys/sysctl.h
	stropts.h		
	net/if.h		
	arpa/inet.h		
	sgtty.h		
	sys/mman.h		
	sys/socket.h		
	sys/utsname.h		
	sys/vlimit.h		
	sys/vtimes.h		
	sys/wait.h		
	syslog.h		
	unistd.h		

Headers using the file system, networking, or other IO.

Even when a specification is necessarily weak, there may still be some properties which can be modeled. For example, many functions have a distinct success case or failure case. This usually takes the form of the knowledge that either some error flag is set, or a well-structured result is returned. Additionally, the results are limited to rational possibilities. For example, the function

‘cos’ will always return between -1 and 1. For many functions, error and exceptional cases are also described. This is common in the mathematical functions since those cases can often be easily inferred from the parameters. However, error cases are also described in some networking functions such as ‘arpa/inet.h’.

5.2.3 Difficult

Some functions can be fully reasoned about, but require more complexity in the models. Strings are one example of this, where we leverage models provided by Frama-C. Unfortunately, the increased complexity that comes with these models becomes too difficult for automated solvers. In such cases, properties about the string are provided, such as length. Moreover, the exact contents of the string are omitted. Headers such as fnmatch.h, libgen.h, and getopt.h which require string comparisons or manipulations are given weaker specifications so they can be used with an automated solver.

Another instance where a model is required is in the functions that manipulate the locale. However, these models have not been implemented, causing all functions which are locale-aware to be weakened. With these functions, we still provide the frame condition, pre-condition, and error cases when applicable.

Additionally, the memory model used in WP can cause limitations. We generally use the Typed memory model to ensure soundness. However, some functions, such as those in search.h, expect a struct argument that is larger than their type declaration. This requires the Typed+cast model be employed in order to reason about those functions.

5.2.4 Non-Behavioral

The majority of available specifications only provide non-behavioral reasoning. In ACSL, this translates to a well-defined precondition and an assigns clause. However, the post condition is usually weak or completely omitted, with the exception of the assigns and simple properties such as non-nullness, known ranges of numeric output, and validity of pointers and opaque types. Because these specifications are on libraries, this is an effective approach to ensure proper use of the library functions.

The header assert.h can terminate the program using the idioms described in section 4.7. These functions are difficult to use in practice because they must pessimistically assume termination. Moreover, validation of the specifications will fail.

5.2.5 Improvable

While many of the functions described above have weak specifications from inherent limitations, some can be improved with the current technologies. Some will require additional logical modeling similar to those described in the section 5.3, including state tracking and expanding the definitions from ctype.h for wide characters. Others will have reasonable results defined.

crypt.h	sys/times.h
regex.h	sys/timex.h
strings.h	time.h
sys/auxv.h	wchar.h

sys/sysinfo.h	wctype.h
sys/time.h	wordexp.h

Headers that can be immediately improved through future work.

Files which have not yet been implemented are provided as header files with no contents. This allows them to be included, but using a function will result in a compile time error. Future work will be done, first to add the function headers, and then provided specifications.

linux/fs.h	netinet/ip_cmp.h
linux/if_addr.h	netinet/ip.h
linux/if_netlink.h	netinet/in.h
linux/netlink.h	sys/statfs.h
linux/rtnetlink.h	uchar.h
netinet/in_system.h	ucontext.h

Headers that are unimplemented.

5.3 Strong specifications

In specific cases, complete specifications can be provided. First, functions with a guaranteed result such as in error.h and err.h, which guarantee termination, can be trivially given a complete specification. Additionally, some mathematical functions can have precise specifications because arithmetic is built in to the logical system. Finally, complete specifications are also provided when strong models are also provided, such as in string.h, where the functions are logically described as shown in section 4.5. More complex strong models can have a downside in that automated reasoning can take excessively long times, becoming unusable depending on the circumstance.

Logical predicates and functions were defined for ease of reasoning, and to track the state of an external interaction as described in section 4.5. For example, the header ‘ctype.h’ introduces predicates for character sets as defined by the ISO standard. This not only makes the specifications human readable, but also provides a mechanism for developers to describe characters in a high level way.

POSIX	dirent.h	dlfcn.h	glob.h	iconv.h	semaphore.h	netdb.h	nl_types.h	ndbm.h

Functions that track an external state.

Some header files do not define any functions. Moreover, they do not contain any specifications, since we are primarily providing function contracts. These headers typically define pre-processor macros, or type definitions.

byteswap.h	float.h	sys/param.h
cpio.h	iso646.h	sys/syscall.h
elf.h	limits.h	sys/types.h
endian.h	stdbool.h	sys/un.h
errno.h	stddef.h	sys/vfs.h
features.h	stdint.h	tar.h

Headers that have no functions defined by their standard.

5.4 Unsupported by frama-c

Some features are unsupported in Frama-C or WP. When a header file is dedicated to one of these features, the program cannot be soundly reasoned about. These features are described in section 3.

Header	Unsupported Feature
complex.h	Complex Numbers
execinfo.h	Concurrency
tgmath.h	Complex Numbers
fenv.h	Unsupported Environment
pthread.h	Concurrency

Headers that cannot be included.

6 Conclusion

The work of this project has resulted in three broadly useful results:

- A significantly expanded set of specifications for commonly used C library functions
- Documentation and reports of bugs and limitations in using Frama-C
- A set of common practices that helped make annotation writing uniform across our work group.

The resulting annotations were essential in other parts of our project and in companion projects that also used ACSL and Frama-C. As the work products of an NSF-funded project, the enhanced specifications are available for public use. Any users are encouraged to respond with comments about their experiences using these specifications or adding to them.