Contracts in OpenBSD

MSc. Dissertation Report

Murat Torlakcik

A thesis submitted in part fulfilment of the degree of MSc Advanced Software Engineering in Computer Science with the supervision of Dr. Joseph Kiniry.

School of Computer Science and Informatics

University College Dublin

11 April 2010

Table of Contents

Abstract

OpenBSD is widely regarded as the world's most secure operating system. It represents the combined effort of hundreds of dedicated individuals, most of who work in their spare time for free, over nearly a decade.

OpenBSD developers try very hard to write code that is well-designed, clean, maintainable, but above all, secure. A number of techniques are used for this, but most of them are relatively ad hoc when compared to the kind of program development that KindSoftware research group regularly participates in [1].

One way to further improve the quality and security of OpenBSD code is to use various types of static analysis to reason about the code to (perhaps automatically) ensure that it conforms to some form of (perhaps lightweight or entirely implicit) specification.

Unfortunately but predictably, the vast majority of OpenBSD is written in C. Historically, API functions in C are documented via standards (like POSIX), manual pages, and code comments. And while some of these "informal specifications" were written by (possibly large) groups of (very smart) people, they are frequently imprecise, incorrect, and difficult to maintain.

The goal of this project was to learn more about writing useful documentation, primarily in the form of contracts, for C program code in OpenBSD. A small number of functions were chosen for examination based upon their size, complexity, and utility from various C libraries and the OpenBSD kernel.

Each of these functions had several specifications written: one based upon the standard, one based upon the manual page and finally one based upon the program code and the derived "correct model" for the function. By comparing these specifications, it was determined if/where each informal specification was incorrect and revisions were proposed to improve clarity and correctness. Additionally, it was attempted to use the Frama-C framework and its Jessie plug-in on the annotated functions to check their correctness with respect to those specifications. All the documentation was reconsidered in the light of the new contracts.

Acknowledgements

The author would like to thank his supervisor Dr. Joseph Kiniry for his help, guidance and his time.

The author would also like to thank his wife, Suzanne, for her understanding and her support.

1 Introduction

Today, it is beyond doubt that static analysis tools detect defects in software. Static analysis tools evaluate software without running it but rather in the abstract and look for violations of recommended practices and defects. Static analysis tools have been proven to find important defects including security defects in software [2].

The use of static analysis tools range from code metrics tools to heuristic bug finding tools. The heuristic bug finding tools provide a more detailed inspection of the source code and try to detect possible bugs [3].

Another use of static analysis is the verification of properties of software by static reasoning. This is done by adding specifications to program parts to document the interfaces and proving that the implementation satisfies these specifications in possible paths of execution. These specifications record the intended behaviour of the program and can prove to be crucial for programming in terms of code maintenance and testing correctness. The code alone is not enough to document the intention of the code [4].

With static verification, the code is not executed; instead the verification is done through static analysis of the code using logical techniques to prove that the specifications will hold at runtime. This generally requires fairly complete annotations of the code as well as the modules and the libraries the code depends on [5].

Formal methods are used for code verification and each program-specification pair constitutes a correctness theorem which the code verification tries to prove. Under certain conditions described, the program should exhibit the behaviour described by its documentation. If a theorem is not proven, code verification can also help understand why it cannot be proven [6].

By using formal methods, software quality can be improved significantly and several mathematical and tool infrastructures for automatic and interactive verification have been developed through research in semantics, specification and verification [7].

For the work presented in this paper; such a tool, Frama-C Framework [3], was used. Using this framework, it is possible to verify that formal specifications hold for the source code they are provided for. The provided specifications can be partial and focus on one aspect of the program at a time. ANSI C Specification Language (ACSL), which is a Behavioural Interface Specification Language, can be used to write the specifications [3].

ACSL is based on very popular Java Modelling Language (JML) [8]. Behavioural Interface Specification Languages are used to express the intended behaviour of a program [4].

It is one of the software engineering practices to keep all system artefacts including design, implementation and documentation consistent and in-sync. However, in practice as the implementation changes, the documentation is only updated afterwards and only now and then. As a result, seldom two of the system artefacts are kept in-sync and hardly ever all artefacts are kept in-sync. Design and documentation artefacts end up not reflecting the actual running system but rather earlier designed system [9].

Software documentation should not be maintained separately to the actual code and become an afterthought. It should become more abstract and integral part of the code; and it should be extractable by software tools. This is the Design by Contract's approach where the documentation is not seen as a separate product [10].

Using Design by Contract, a contract is written for each method of a *class* or *interface* and the contracts express what is required from the client and what is ensured in return to the client. A Design by Contract specification is more abstract than code and focuses on the assumptions and the goals to be achieved. This approach provides good documentation which is better than just code alone or informal documentation such as comments or *Javadoc* comments. Formal specification is generally automatically checkable and therefore can help detecting errors before they propagate to the rest of the system. The specifications are also more likely to be kept in-sync with the actual implementation since they can be checked automatically [11].

1.1 Project Specification

The goal of this project was to learn more about writing useful documentation, primarily in the form of contracts, for C program code in OpenBSD [12]. OpenBSD is widely regarded as the world's most secure operating system. It represents the combined effort of hundreds of dedicated individuals, most of who work in their spare time for free, over nearly a decade. OpenBSD developers try very hard to write code that is well-designed, clean, maintainable, but above all, secure. A number of techniques are used for this, but most of them are relatively ad hoc when compared to the kind of program development that KindSoftware research group [1] regularly participates in.

One way to further improve the quality and security of OpenBSD code is to use various types of static analysis to reason about the code to (perhaps automatically) ensure that it conforms to some form of (perhaps lightweight or entirely implicit) specification. Unfortunately but predictably, the vast majority of OpenBSD is written in C. Historically, API functions in C are documented via standards (such as POSIX), manual pages, and code comments. And while some of these "informal specifications" were written by (possibly large) groups of (very smart) people, they are frequently imprecise, incorrect and difficult to maintain.

A small number of functions were chosen for examination and analysis using Frama-C [3] based upon their size, complexity, and utility and the tool limitations. These functions come from string C libraries in the OpenBSD kernel. For each of these functions, several specifications were written: one based upon the standard, one based upon the manual page and finally one based upon the program code and the derived "correct model" for the function. By comparing these specifications it was attempted to determine if/where each informal specification was incorrect and revisions were proposed to improve clarity and correctness. It was also planned to get the formal contracts incorporated into the source code and manual pages for OpenBSD.

It was also attempted to use the Frama-C framework and its Jessie plug-in on the annotated functions to check their correctness with respect to those specifications and to reconsider all the documentation in the light of the new contracts.

1.2 Roadmap

Chapter 2 provides the background information. First three sections of this chapter discuss the static analysis of C programs and why it is important; the lightweight static checkers in general and the C annotation languages. The remainder of this chapter discusses more specific tools and concepts related to this project. Sections 2.4 and 2.5 introduce Why platform [13] and the Frama-C Framework [3] respectively. The Frama-C Framework is discussed in more detail in terms of its architecture including CIL [14] and Jessie intermediate languages [15]; and its plug-in architecture. Finally, Section 2.6 gives a short overview of OpenBSD and why it was chosen for this project.

Chapter 3 discusses the first phase of the project which includes the preparatory work done and the triaging of OpenBSD code to choose functions to examine and write contracts for. Section 3.3 discusses how the correctness of the final specifications written was checked using Frama-C Framework.

Chapter 4 discusses the contracts written for the string functions in OpenBSD.

2 Background

2.1 Static Analysis of C Programs

The C programming language is a general-purpose programming language and it was developed in 1972 [16].

C is important in terms of static analysis as despite being a fairly old language, C is one of the most popular languages. Applications written in C are widely used by different industries and some of these industries critically depend on software [15].

One of the characteristic of C programming language that makes it quite popular is its flexibility to deal with low-level constructs. However, this flexibility also makes it difficult to understand and analyze automatically. There are not many tools to analyse C programs as many of its features are not easy to reason about [14].

One of the complications of analysing C programs is the fact that the language does not provide a mechanism for aliasing which stays implicit in a C program. As there is no strong typing in C; given limited information, analysis has to assume that any two pointers may alias [15].

C programming language also has various language issues. While most high-level programming languages guarantee memory safety, C language does not protect against memory errors. This is caused by its low-level orientation and leaves the programs prone to dangerous forms of safety violations including buffer overflows. One of the reasons also for buffer overflow errors is that C does not provide a mechanism for size of a buffer at runtime. Without support from the language, the programmer solely is responsible for valid memory access by taking all possible cases for input into account. Otherwise, improper input can leave the programs vulnerable to attacks. No technique can completely enforce memory safety given the range of possible exploits. Therefore, the memory safety becomes the most important property of C programs in terms of ensuring safety [15].

The language issues, the fact that C is difficult to analyze and the high number of critical C programs in industry make analysis of C programs very important.

The first widely used static analysis tool for C language is considered to be the lint tool. Since the first use of this rather limited tool, there has been important research work in the area, driven mostly by security vulnerability concerns. This research work and the commercial industry have produced much more advanced static analysis tools [2].

2.2 Lightweight static checkers

Using static analysis, there are limits to what can be analysed. Static analysis tools designers are challenged with tradeoffs between required effort and analysis complexity and also between precision and scalability. Certain comprises are often made to achieve faster analysis that scales to larger programs [17].

Given the tradeoffs, there is a range of static analysis tools. Some identify potential defects like variable use before definition and possible out of range expressions. Heuristic tools like the lint tool falls in this category and produce some noise and false positives as they use formally unsound analytical techniques [6].

At the opposite side of the range, one can found the program verifiers that require formal specifications and use automated theorem provers to prove properties of a program. However, as a result, program verifiers and the techniques used can be too expensive and heavy [17].

Using lightweight static analysis techniques requires more incremental effort but this is nothing compared to the effort required for full program verification. This is achieved by certain compromises that sacrifice soundness and completeness. The design of lightweight analysis tools forego theoretical claims and favour useful results by using heuristics to assist the analysis. The goal is to achieve useful tools for producing useful results with reasonable effort even if this means these tools will sometimes generate false positives and sometimes even false negatives with imprecise results that miss real problems. Therefore, lightweight static analysis tools like Splint can be configured to suppress certain warnings and allows weaker or stronger assumptions for checking [17].

Splint is a good example of a lightweight static analysis tool and is explored further in the following section.

Using Extended Static Checking for automatic verification has been successful in the industry and academia as it is lightweight and easy to use and it is more reasonable to expect typical programmers to use automatic verification than interactive verification. Although interactive verification technology is generally complete and sound, it requires mathematical and practical expertise [7].

Prefast is another example of a lightweight static analysis tool that is designed to run quickly by sacrificing in-depth analysis and generating some noise [18].

2.2.1 Splint

Splint is a static analysis tool for C programs and checks for programming mistakes and security issues. Splint is a successor to Lint and LCLint tools with the focus on detecting security vulnerabilities. Its name is short for “Specification Lint” and “Secure Programming Lint”. Splint tool is developed and maintained by the Secure Programming Group at the University of Virginia [19].

Splint, being a successor to Lint and LCLint, provides many of the traditional lint checks but by using annotations in the source code, it also improves these checks and also provides more powerful checks. Splint tool is flexible and allows programmers to configure the tool to select classes of errors for each project. The number of defects that are detected by Splint can be increased by turning on different checks and providing more information in code annotations. Splint is also extensible and allows defining new annotations and associated checks [19].

Splint annotations are further discussed in Section 2.3.3.

Using Splint to detect and fix errors can be done by following an iterative process where Splint is run repeatedly until all the issues are fixed. Splint is very fast, it checks approximately 1000 lines of code per second so this iterative process is reasonable. After each run of the tool, either the code or the annotations are fixed to deal with the warning messages. Splint is invoked by passing the list of files to check [17]:

```
splint *.c
```

It is also possible to turn off reporting for some of the warnings by selecting weaker checking:

```
splint -weak *.c
```

Few hundred flags are provided to control checking and message reporting. The information on how to turn off a specific check is also provided. A typical warning message is as follows [17]:

```
sample.c: (in function faucet)
sample.c:11:12: Fresh storage x not released before return
A memory leak has been detected. Storage allocated locally is not released
before the last reference to it is lost. (Use -mustfreefresh to inhibit
warning)
sample.c:5:47: Fresh storage x allocated
```

Splint generates pre-conditions and post-conditions using the internal rules or the annotations. It analyses a function using the annotated preconditions and verifies that the implementation ensures the post-conditions. For example, the following declaration generates the post-conditions *maxSet(buf) = MAXSIZE – 1* and *minSet(buf) = 0* [17]:

```
char buf[MAXSIZE]
```

If in the code, the expression *buf[i]* is used as an *lvalue*, Splint generates the pre-condition *maxSet(buf) >= i*. Splint resolves pre-conditions using post-conditions from the previous statements and for a function from any annotated pre-conditions. A warning is generated if a generated pre-condition cannot be resolved or a post-condition cannot be satisfied. Going back to the example above, if Splint cannot determine that the value of variable *i* is between 0 and *MAXSIZE – 1* then it will generate a warning message [17].

Splint uses constraint-specific algebraic rules to simplify constraints and uses axiomatic semantics to propagate constraint across statements. Splint also uses heuristics to recognize common loop forms in order to handle loops so it does not use loop invariants [17].

2.3 C annotation languages

Highly expressive notations are required to describe the behaviour and properties of complex software for use by verification or static analysis tools. Formal specification languages provide mathematically unambiguous notations for describing the behaviour and properties of software and help verify that these properties are satisfied by the implementation. These specification languages are designed to be expressive enough, easy to use and automatically verifiable [4].

This section discusses C annotation languages that are used to add specifications to C programs. Standard Annotation Language (SAL) is an example of a C annotation language that is used to express additional specifications like the expected buffer size. Checking that these annotations are respected is done by a dedicated compiler. Although the annotation languages like SAL generally cannot completely express all required properties for safety, SAL is being used at Microsoft to accomplish greater degrees of dependable software [15].

The Prefast tool, like other lightweight static analysis tools, sacrifices in-depth analysis for performance. This is where the SAL annotations come in to help Prefast with the accuracy of the static analysis by expressing properties, like requirements of functions and use of parameters, formally. As well as reducing the false positives, using SAL annotations and static analysis tools like Prefast helps deepening the analysis of specified behaviours. This is a good approach to increase security and reduce the number of defects in C programs [18].

2.3.1 ACSL

This section describes the basics of ACSL which is an acronym for ANSI C Specification Language. ACSL has been used for the contracts in this project so Chapter 4 will discuss this language in more detail.

ACSL is a Behavioural Interface Specification Language that is used to describe behavioural properties of C programs for deductive verification. ACSL was inspired by Caduceus tool's specification language [13] which itself was inspired by the Java Modelling Language [8] that uses ESC/Java2 tool [7] for static and deductive verification [20].

Frama-C Framework [3] provides an ACSL implementation and it adds more features of ACSL at every release. For this project, the version 1.4 of ACSL design was used. Some of the more advanced features of ACSL provided by the Frama-C Framework for this version are experimental but the core features of ACSL required for basic use of the specification language are provided [20].

An important design choice of ACSL is that the semantics of ACSL logic expressions is based on first-order logic and therefore expressions in ACSL are never undefined [20].

Although ACSL is inspired by JML, there are important differences between the two. While ACSL targets C, JML targets Java. While C is a low-level structured language, Java is a high-level, object-oriented language; consequently, the way the behaviours are specified is different. ACSL allows all first-order logic formulas and when theorems cannot be proven it depends on manual deductive verification using an interactive theorem prover. JML depends on runtime assertion checking when static analysis and automatic verification fails [20].

2.3.1.1 Annotations

ACSL specifications are added directly in the C source files in the form of annotations in comments that start as /*@ for multi-line comments or //@ for single line comments. As the specifications are added inside the comments, the compilation of the source code is not affected [20].

The following are the kinds of global annotations that can be used [20]:

- Function contract that is added before a function declaration or definition.
- Type invariant that allows annotating type names introduced by *typedef* and express invariants.
- Logic specifications that are added at global declaration level and define new logic types, logic functions and predicates with axioms.

Statement annotations that can be used are as follows [20]:

- *Assert* clauses can be used anywhere that a C label is allowed.
- Loop annotations (*invariant*, *variant*, *assign* clauses) are used before a loop statement.
- Statement contracts can be added before a statement or a block and are similar to the function contracts.

- Ghost code is regular C code that is only visible from the specification and can only modify ghost variables.

2.3.1.2 Function Contracts

A simple function contract has simple clauses and has no named behaviours:

```
/*@ requires P1; requires P2; ...
  @ assigns L1 assigns L2; ...
  @ ensures E1; ensures E2; ...
  @*/
```

The semantics of such a contract is as follows [20]:

- The caller has to guarantee a state where the property *P1 && P2 && ...* holds. If nothing is specified, then *\true* is used as default.

- After the call, the callee guarantees a state where the property *E1 && E2 && ...* holds. If nothing is specified, then *\true* is used as default.

- The function does not modify any memory location of the pre-state that outside the set *L1 [L2 [: : :* which is interpreted in the pre-state. If nothing is specified, then the caller cannot be sure of the side-effects of the function.

2.3.1.3 Loop Invariants

A simple loop annotation can be of the form as follows:

```
/*@ loop invariant I;
  @ loop assigns L;
  @*/.
```

The semantics of such a contract is as follows [20]:

- Before the loop is entered, the predicate *I* is required to hold.

- If the loop ends normally, the predicate *I* is preserved as follows:

  - For a *while* loop as below, predicate *I* must be preserved by the side-effects of *c* followed by *s*:

    ```
    while (c)
      s;
    ```

  - For a *for* loop as below, predicate *I* must be preserved by the side-effects of *c* followed by *s* followed by step:

    ```
    for (init; c; step);
    ```

  - For a *do-while* loop as below, predicate *I* must be preserved by *s* followed by the side-effects of *c*:

    ```
    do
     s;
    while (c);
    ```

- The predicate *I* is an inductive invariant, that is if *I* is assumed *true* in some state where the condition *c* is also *true*, and if execution of the loop body in that state ends normally at the end of the body or with a continue statement, *I* is true in the resulting suitable way:

    - for a *while (c) s* loop, *I* must be preserved by the side-effects of *c* followed by *s*;
    - for a *for (init;c;step) s* loop, *I* must be preserved by the side-effects of *c* followed by *s* followed by *step*;
    - for a *do s while (c);* loop, *I* must be preserved by *s* followed by the side-effects of *c*.

- During the loop iteration, the loop only modifies locations that are members of the set *L*.
- If the loop exits early as a result of a *break*, *goto* or a *return* statement, then the loop invariant is not required to hold.

2.3.1.4 Logic Labels

Logic labels are used in statement annotations to refer to a value of an expression in a given state [20]:

```
\at(e,id)
```

Above, *e* denotes the expression and *id* denotes the state label. Four predefined logic labels are described as follows [20]:

- *Here* can be used in statement annotations and refers to the state where the annotation is present.
- *Old* can be used in *assigns* and *ensures* clauses and refers to the pre-state.
- *Pre* label can be used in statement annotations and refers to the pre-state of the function.
- *Post* can be used in *assigns* and *ensures* clauses and refers to the post-state.

The construct *\old(e)* is a short-hand for *\at(e,Old)*.

For examples of ACSL annotations, see Chapter 4.

2.3.2 SAL

Microsoft's Standard Source Code Annotation Language (SAL) provides annotations to specify a function's behaviour and the way it uses its parameters. The SAL annotations are added in the source code before a function's parameters and the return value [21].

Main benefit of using SAL annotations is that more bugs can be identified easily with some effort on annotating functions. SAL helps static analysis tools to detect code defects and security issues in C/C++ code at compile time by specifying more contextual information on the functions. Issues identified by SAL and Prefast are usually real bugs [22].

The functions that read or write to buffers are the main focus of SAL meta-language. SAL annotations help define proper use of buffers which is hard to ensure in C programming language given the ambiguous nature of pointers. So that this benefit is leveraged, Microsoft adds annotations to most code; C runtime functions included with Visual Studio and Windows SDK code are mostly annotated. This means that even if the client code calling these functions is not annotated, bugs can still be found [22].

SAL annotations can be grouped into two classes. First class of annotations is buffer annotations that specify how a function uses its pointer parameters. The second class is the advanced annotations that specify inexpressible properties or more complex buffer behaviour [21].

Although SAL annotations look different in style to ACSL annotations as they are added as properties on the function parameters, ACSL annotations can also be expressed in a similar way by defining annotations using *__declspec* construct [3].

2.3.2.1 Annotations

SAL annotations can be combined together (of course not all combinations are valid). Combinations of the following macros are the most common:

Indirection Description	deref, deref_opt
Direction Description	in, out, inout
Buffer size units Description	ecount, bcount
Initialization Description	full, part
Required or optional buffer Description	opt
Buffer size Description	size, (size, length)

These macros expand to low-level annotations. For example, *__in* expands to following annotations:

```
pre valid pre deref readonly
```

2.3.2.2 Example 1

```
void  func(__in int *i, __out int *o);
```

Here, the function will only read from the buffer *i*. The caller must provide the buffer *i* and initialize it. The pointer to the buffer *i* must not be null. The function will only write to the buffer *o*. The caller must provide the buffer *o*, and the function will initialize it. The pointer to the buffer *o* must not be null.

2.3.2.3 Example 2

```
void func(__in_opt int *i);
```

The pointer to the buffer might be null and therefore will be tested before being de-referenced. The function will only read from the buffer if provided and in that case the caller must initialize it.

2.3.2.4 Example 3

```
void func(__inout_opt int *o);
```

The function may read and write to the buffer that must be provided and initialized by the caller. The pointer to the buffer might be null and therefore, the function will need to check it before de-referencing it.

2.3.2.5 Example 4

```
void func(__in_ecount(size) char *t, int size);
```

The total buffer size is given as an explicit element count which is tied to the last parameter.

2.3.2.6 Example 5

```
void func(__out_ecount(size) char *t, int size);
```

The function will initialize and write to the buffer that must be provided by the caller. The total buffer size is given as an explicit element count which is tied to the last parameter.

2.3.2.7 Example 6

```
void func(__deref_in int **ppi, __deref_out int **ppo);
```

The function will only read from the buffer **ppi* provided and initialized by the caller and it must not be null. The function will write to the buffer **ppo* provided by the caller. The function will initialize buffer **ppo*. Both buffers are one element long as no size information is given.

2.3.2.8 Example 7

```
void func(__deref_opt_in int **ppi);
```

The function will read from the buffer **ppi* if provided. If the pointer *ppi* is null, then the rest of the annotation is ignored.

2.3.3 Splint's Annotation Language

Efficient and scalable analysis done by Splint tool can be deepened to detect even more implementation flaws by adding annotations in the form of stylized comments to the programs. These comments provide additional information about types, variables or functions and improve checking in general as well as checking that the implementation is consistent with the annotations [2].

Similar to ACSL, the annotations are added in stylized comments that begin with /*@ although character @ can also be changed to any printable character by using *-commentchar <char>*. One difference to ACSL and SAL is that many of the Splint's annotations control settings for checks.

Splint also allows new checks and annotations to be defined. This allows detecting new vulnerabilities or application specific property violations [2].

Some examples are given below from the Splint tutorial [19].

2.3.3.1 Example 1

Running Splint for the example code below produces a warning for the first function about possible null pointer dereferencing. As the second function checks if the pointer is null before dereferencing it, no error is reported for that function.

```
char func1 (/*@null@*/ char *s)
{
  return *s;
}
char func2 (/*@null@*/ char *s)
{
  if (s != NULL)
    return *s;
  return ‘\0’;
}
```

2.3.3.2 Example 2

To illustrate *nullwhentrue* annotation, the second function above can be re-written as follows:

```
  /*@nullwhentrue@*/
  bool isPointerNull (/*@null@*/ char *p)
  {
    return (p == NULL);
  }

  char func2 (/*@null@*/ char *s)
  {
    if (!isPointerNull(s))
      return *s;
    return '\0';
  }
```

No errors will be reported as using the *nullwhentrue* annotation means that *s* must not be null when the statement that dereferences pointer *s* is reached.

2.3.3.3 Example 3

For expressing pre-conditions and post-conditions, *requires* and *ensures* clauses can be used for functions. Function declarations can have multiple *requires* and *ensures* clauses as long as they are not contradictory. While Splint is checking a function implementation, if it finds that there is an execution path where the return state does not satisfy the post-condition, it issues a warning message.

In the following example, the annotation expresses a safety property that the buffer *s1* should be big enough to hold buffer *s2* as a pre-condition:

```
  /*@requires maxSet(s1)>= maxRead(s2) @*/
  void func(char *s1, char *s2);
```

The annotations *maxSet* and *maxRead* are buffer attribute annotations. Splint uses these two properties to model blocks of contiguous memory. The *maxSet* annotation annotates a buffer and denotes the highest address that be safely used as an *lvalue* and *maxRead* denotes the highest index of a buffer that can be accessed safely. For example, for the following declaration:

```
  char buf[MAXSIZE];
```

maxSet(buf) will be equal to *MAXSIZE – 1*. Splint will issue a warning at the call site for this function if it cannot prove that *maxSet(s)>=maxRead(t)* holds.

A related annotation is the *nullterminated* annotation which annotates a buffer as a null-terminated string.

2.3.3.4 Example 4

```
  /*@warn bufferoverflowhigh “Custom warning message.“@*/
```

In the example above, a potentially dangerous function is annotated to warn the programmer about buffer overflow risk. This warning will only be issued if the *bufferoverflowhigh* flag is set.

2.3.3.5 Example 5

It is possible to annotate an abstract type with *mutable* or *immutable*:

```
typedef /*@abstract@*/ /*@mutable@*/ char *aString;
typedef /*@abstract@*/ /*@immutable@*/ struct aStruct;
```

This is useful in terms of expressing assignment semantics and sharing semantics. If a mutable type is implemented without sharing semantics, then Splint issues a warning.

2.3.3.6 Example 6

Using Splint annotations, object lifetime assumptions can also be described. The *only* annotation suggests a requirement to release storage. For example, allocator standard library function *malloc* is annotated using *only* annotation on return value:

```
/*@only@*/ /*@null@*/ void *malloc (...);
```

And function free only takes an *only* parameter which must reference an unshared object:

```
void free (/*@only@*/ /*@out@*/ /*@null@*/ void *ptr);
```

Therefore, in order to satisfy the requirement to release an allocation returned by *malloc* is to pass it to function *free*. Splint will help track memory leaks using the annotations and issue warning for any code path that does not satisfy these requirements.

Also, notice the use of *out* annotation which is similar to SAL annotations.

2.3.3.7 Example 7

```
/*@noreturn@*/ void exit();
```

The function above is annotated and described as a function that never returns. Annotations like *noreturn* allows Splint to know about control flow behaviour and analyse code correctly. Similarly, *maynotreturn* annotation can be used for functions that usually return normally. However, it is also possible to use more precise annotations:

```
/*@noreturnwhenfalse@*/ void assert (...);
```

This allows Splint to analyse code like the following correctly:

```
assert (p);
*p = 0;
```

2.4 Why Platform

Why is a platform for software verification and contains the following tools [13]:

- Why which is a general purpose verification condition generator.
- Krakatoa tool for verifying Java programs.
- Caduceus tool for verifying C programs.

Why Platform can be integrated with many of the existing theorem provers including Simplify, Alt-Ergo, Yices and Z3 [13].

Why Platform is used for proving advanced behavioural specifications. Using the Why Platform, Java and C code is translated into Why programs which are sets of functions with pre-conditions and post-conditions. First, Krakatoa and Caduceus tools interpret the annotations added to the source code. The Krakatoa tool interprets annotations written using Java Modelling Language and Caduceus tool interprets annotations written in its own specification language which was inspired from Java Modelling Language. Once the annotations are interpreted, Why generates verification conditions which are proven using one or multiple theorem provers. A verification condition can be described as a logical formula that is valid if the code satisfies the given annotation. Why Platform generates verification conditions using Weakest Precondition Calculus. It also generates some safety verification conditions automatically [23].

Caduceus tool is now obsolete and Frama-C Framework, which is discussed in the following section, should be used instead [13].

2.5 Frama-C Framework

Frama-C, which stands for Framework for Modular Analysis of C programs, is open source software written in Ocaml by Commissariat à l'Énergie Atomique and Inria [16].

Frama-C is a framework where several static analysers for C language can collaborate and build on results of other analysers. This design allows Frama-C to accomplish more sophisticated tasks. Frama-C can be used for [3]:

- Observing possible variable value sets at different program execution points.
- Slicing a C program in order to simplify it. This is especially useful for understanding other people's code.
- Navigating the program data flow.
- Verifying formal specifications written in ACSL for C code.

2.5.1 The plug-in architecture of Frama-C

Frama-C is extensible and uses a plug-in architecture. Frama-C kernel centralises the analysis and provides interfaces that the plug-ins can use to interact with other plug-ins. Several plug-ins are included with the framework and new ones can be added to extend functionality with less effort by using existing plug-ins [3].

In next two sections, two existing plug-ins, Value and Jessie plug-ins will be discussed.

2.5.1.1 Value Plug-in

The Value plug-in is used for value analysis of a C program based on Abstract Interpretation. It automatically computes possible variable value sets for a program. Using these value sets, it is possible to infer that there are no runtime errors. Warnings are issued for any operation in code that can lead to a runtime error. The plug-in guarantees that no warning for an operation means that the operation cannot be the cause of a runtime error. Although at the time of the writing there are known limitations of the plug-in; it can only analyse sequential code and it can only analyse simple dynamic allocations correctly [3].

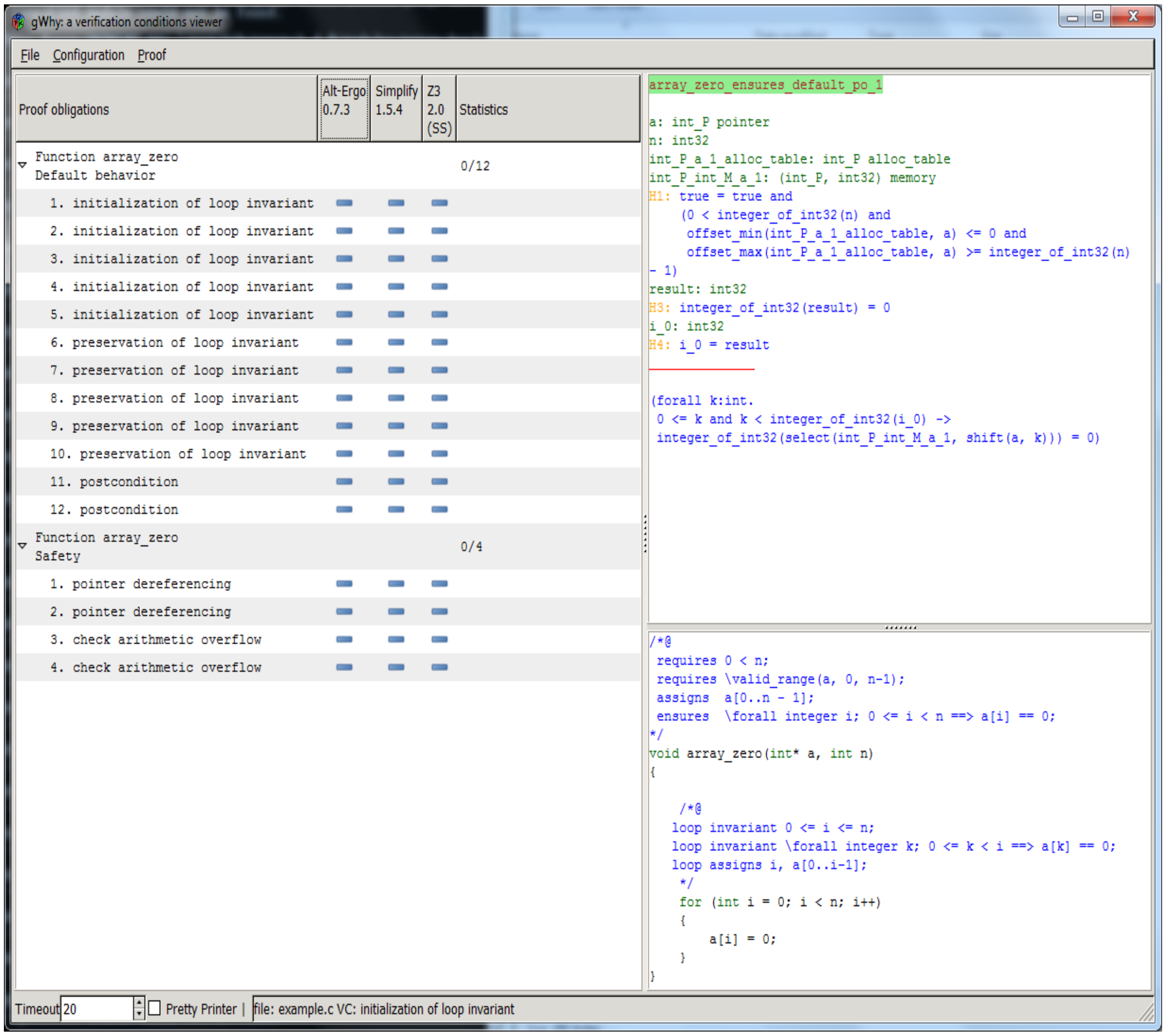
The value plug-in can be used in either batch mode or with Frama-C's graphical user interface. When used with the graphical user interface, at each point of the program, the inferred sets of values are displayed. As the Frama-C Framework is designed as a collaborative tool, the results of value plug-in can also be used by other plug-ins [3].

The plug-in is run as follows and some examples of runtime errors that value analysis can detect are divisions by zero, invalid pointer access and out of bound buffer accesses [3]:

```
frama-c -val program.c
```

2.5.1.2 Jessie Plug-in

The Jessie plug-in for Frama-C Framework is used for deductive verification of C code annotated with ACSL. The plug-in uses the Why Platform tools internally to generate the verification conditions which can be proven by automatic theorem provers. The theorem provers are external to Frama-C and many theorem provers such as Simplify, Z3 and Alt-Ergo can be used with Frama-C. Interactive theorem provers can be also be used if needed [3].



Proof obligations	Alt-Ergo 0.7.3	Simplify 1.5.4	Z3 2.0 (SS)	Statistics
Function array_zero Default behavior				0/12
1. initialization of loop invariant	▬	▬	▬	
2. initialization of loop invariant	▬	▬	▬	
3. initialization of loop invariant	▬	▬	▬	
4. initialization of loop invariant	▬	▬	▬	
5. initialization of loop invariant	▬	▬	▬	
6. preservation of loop invariant	▬	▬	▬	
7. preservation of loop invariant	▬	▬	▬	
8. preservation of loop invariant	▬	▬	▬	
9. preservation of loop invariant	▬	▬	▬	
10. preservation of loop invariant	▬	▬	▬	
11. postcondition	▬	▬	▬	
12. postcondition	▬	▬	▬	
Function array_zero Safety				0/4
1. pointer dereferencing	▬	▬	▬	
2. pointer dereferencing	▬	▬	▬	
3. check arithmetic overflow	▬	▬	▬	
4. check arithmetic overflow	▬	▬	▬	

```
array_zero_ensures_default_po_1

a: int_P pointer
n: int32
int_P_a_1_alloc_table: int_P alloc_table
int_P_int_M_a_1: (int_P, int32) memory
H1: true = true and
    (0 < integer_of_int32(n) and
     offset_min(int_P_a_1_alloc_table, a) <= 0 and
     offset_max(int_P_a_1_alloc_table, a) >= integer_of_int32(n)
- 1)
result: int32
H3: integer_of_int32(result) = 0
i_0: int32
H4: i_0 = result


(forall k:int.
 0 <= k and k < integer_of_int32(i_0) ->
 integer_of_int32(select(int_P_int_M_a_1, shift(a, k))) = 0)
```

```
/*@
 requires 0 < n;
 requires \valid_range(a, 0, n-1);
 assigns  a[0..n - 1];
 ensures  \forall integer i; 0 <= i < n ==> a[i] == 0;
*/
void array_zero(int* a, int n)
{


    /*@
   loop invariant 0 <= i <= n;
   loop invariant \forall integer k; 0 <= k < i ==> a[k] == 0;
   loop assigns i, a[0..i-1];
    */
    for (int i = 0; i < n; i++)
    {
        a[i] = 0;
    }
}
```

Timeout 20 | Pretty Printer | file: example.c VC: initialization of loop invariant

Figure 1. *Jessie Plug-in and gWhy.*

The plug-in can be launched using the following command line:

```
frama-c -jessie file.c
```

Why Platform's user interface, GWhy, is launched by default. Figure 1 shows the user interface, the generated verification conditions (which are also called proof obligations), and the installed theorem provers which can be run in parallel. A batch mode is also supported using command line options which also include specifying a specific theorem prover [24].

2.5.2 Frama-C Architecture

Figure 2 shows how a C program is translated into a Why program from which the verification conditions are generated using the Frama-C Framework, the Jessie plug-in and the Why Platform [15].

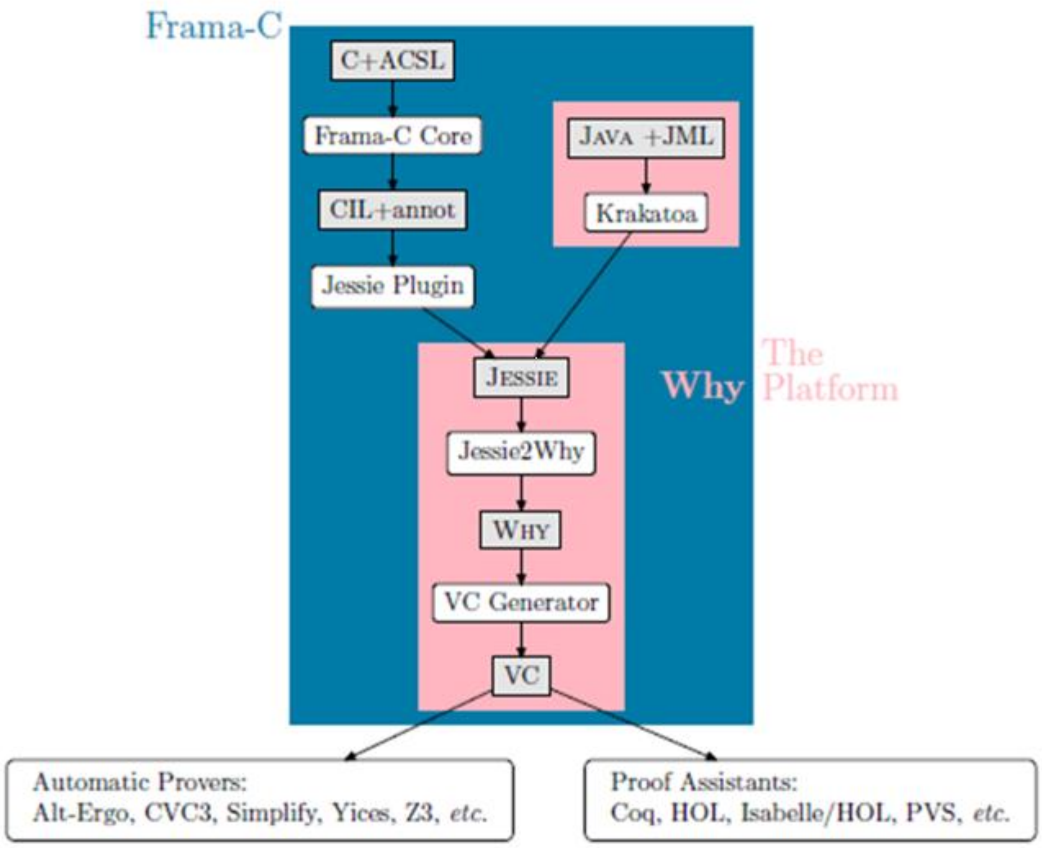


Figure 2. *Flow of translation [15].*

The C program and its ACSL annotations are translated into CIL [14] by Frama-C Framework and then to Jessie intermediate language by the Jessie plug-in. Finally, the Jessie intermediate language program is translated into a Why program which is used by the Why Platform to generate the verifications to submit to theorem provers or proof assistants [15].

2.5.2.1 CIL

C Intermediate Language, CIL, can be used for source-transformation of C code in order to make analysis easier. CIL is a subset of C language; a high-level representation that has fewer, simpler constructs than C language. CIL simplifies a C program by reducing more complicated constructs while staying close to the source and therefore makes it easier to analyse and map conclusions back to the original source program. A front end translates ANSI C and also other C extensions. CIL can also be easily translated back to C [14].

2.5.2.2 Jessie Intermediate Language

Jessie is an intermediate language that acts as a bridge between Frama-C Framework and the Why Platform. As it is an intermediate language, it is not designed to be created by programmers; but rather it is a target for translation from CIL language. The C programs with ACSL annotations are translated into CIL and from CIL they are translated into simpler Jessie programs and Jessie annotations. The goal of using Jessie intermediate language is to generate annotations from which verification conditions can be generated. Jessie constructs are very similar to ACSL and some C constructs such as floating point numbers and function pointers are not supported [15].

2.6 OpenBSD

OpenBSD is a free, BSD-based multi-platform operating system developed by volunteers. The project presented in this paper is based on OpenBSD for the following reasons [12]:

- Strong security. OpenBSD aspires to be the most secure operating system.
- Documentation. OpenBSD provides detailed manual pages for documentation and invests in keeping documentation accurate and up-to-date.

OpenBSD operating system integrates cryptographic software and uses Secure by Default mode where only essential services are enabled. Only two remote holes were found in default install of OpenBSD in more than ten years which is an enviable record. As well as robust and secure software, some of the other goals of the OpenBSD project include [12]:

- To provide the best development platform where users and developers have full access to source code.
- To support different systems and hardware.
- To be developer oriented and politics-free.
- To proactively fix serious problems including security issues.
- To make a CDROM-based release every six months. This is how the project is funded.

OpenBSD is more popular as a server operating system but it is also used as a desktop operating system. It is possible to install additional software including several desktop applications on OpenBSD using *packages* and *ports*. The third party software distributed with OpenBSD is also patched to improve security and code quality [12].

The version that was used for the project is OpenBSD 4.6 and it was installed using the CD-ROM in order to support the project. OpenBSD 4.6 release brings important improvements and new features; however one feature that has to be mentioned is the installation process. Although the installation was already simple and well documented in previous releases, the 4.6 release makes it even simpler and faster.

3 Methodology

This chapter presents the first phase of the project and the preparatory work done. The preparatory work included the installation of the tools, getting familiar with ACSL (See Section 2.3.1) and the Jessie plug-in (See Section 2.5.1.2) by reading tutorials available on the Frama-C Framework website [3], following the public mail list [3] and trying out examples.

The learning phase was followed by the triaging of the OpenBSD code in order to choose functions to write contracts for. The triage process and the methodology followed in the examination of the selected functions are discussed in Section 3.2.

The last section in this chapter discusses the methodology for the specifications and how the correctness of the final versions of the specifications written was checked using the Frama-C Framework.

3.1 Getting the tools ready

The project work required installing and using the following tools:

- Frama-C Framework.
- Why platform with the Jessie plug-in (including gWhy for user interface).
- Theorem provers Alt-Ergo, Simplify and Z3 [26].

Frama-C Framework was in development during this project, so several upgrades were made as the new versions were available. The project started with the Lithium version and finally the specifications were written and verified using Beryllium Second Version. The project was developed on a Windows system using cygwin [25] and gcc [25].

Frama-C Framework is developed in Ocaml [25]. Although, the binary installers for Windows were used at early stages of the project, the tools were compiled and built from the sources as new versions were released.

Prior to the Beryllium version, each Frama-C release included the latest version of Why Platform at that point. Starting with Beryllium versions, Frama-C was no more distributed with Why Platform. Also, starting with Why 2.20 release, Jessie plug-in was included in Why Platform releases and not with Frama-C releases. As a result of these changes, Why Platform was compiled and installed separately to the Frama-C Framework for the project.

The theorem provers are not distributed with Frama-C Framework or Why Platform. Therefore, Alt-Ergo, Simplify and Z3 theorem provers [26] were installed separately. Before the first execution of the Jessie plug-in, it is required to run the *why-config* tool in order to detect the theorem provers.

3.2 Triaging OpenBSD APIs

This section discusses the triage for choosing a small number of functions from the OpenBSD code for examination.

3.2.1 Methodology

Triage was done to pick functions based on their size, complexity, and utility. Rather than doing this manually, it was decided to come up with a process and tools to achieve this. Therefore, it was possible to examine the OpenBSD code base automatically to pick high utility functions based on popularity and criticality to the code base and the tool support.

In order to achieve this goal, several Perl scripts [25] were written and run in a specific order to find most popular functions that did not use the constructs not supported by the Frama-C Framework and its Jessie plug-in.

Frama-C Framework does not have many limitations itself but each plug-in defines its own limitations in terms of C constructs it supports. For example, Jessie does not support pointer casts [3].

The following were the unsupported C features of the Jessie plug-in at the time of the writing [24]:

- Arbitrary *goto*s. Some simpler forward *goto*s are allowed.
- Function pointers.
- Arbitrary casts. There was some experimental support for casting between pointers (except in specifications).
- Union types.
- Variadic functions.
- Floating point computations.

As well as these unsupported C features, also some ACSL features were not supported or partially supported and this contributed to the complexity of the functions and possible annotations to be written for them.

3.2.2 Triage scripts

A multi-step approach was followed and for each step, a Perl script was run that generated a text file which became the input to the following step. The whole process could also easily be fully automated to run all steps sequentially. The scripts ran on a local folder that contained the latest OpenBSD source code. The scripts can be found on the project *Trac* site [25] [27].

The seven scripts and the corresponding triage steps were as follows: *S0_List_Includes.pl*, *S1_List_Funcs_In_Headers.pl*, *S2_List_All_Func_Calls.pl*, *S3_Count_Each_Func_Call.pl*, *S4_Filter_BasedOn_Count.pl*, *S5_List_All_Func_Source.pl*, *S6_Add_Source_For_Func.pl* and *S7_Filter_BasedOn_Support.pl*.

3.2.2.1 Steps 0 - 3

The first step goes through the man files in source tree and filters the *#include* statements so a list of folders is created. The only manual step in the process was at this step to sort and clear the duplicates in this list using a spreadsheet program. Although this could also be automated, it was not done as there were not many duplicates.

The next step goes through the list of folders and lists all the functions in the header files found in these folders so that the following step can count occurences of function calls for each function. The problem here is that going through the list of functions in the text file and counting occurrences for each one in the entire source code base is not feasible. As a reasonable solution, step *S2* goes through all the C source files in the code base once and spits out all the functions for all the function calls found in them.

Using the two generated text files, step *S3* simply counts the occurrences of each function name in the second text file, producing the number of calls for each function.

The output of step 1 generates a file with content similar to the following:

```
e:\openbsd_code/include/arpa/inet.h     inet_nsap_ntoa
e:\openbsd_code/include/blf.h           Blowfish_encipher
e:\openbsd_code/include/bm.h            bm_free
e:\openbsd_code/include/bsd_auth.h      auth_getitem
…
```

The tab separated file lists the header name and the function name on each row. Tab separation makes it easy for the script of the next step to parse the file.

The following snippet from the output of the third step shows the number of times each function was called as detected by the script:

```
54      e:\openbsd_code/include/arpa/inet.h        inet_addr
92      e:\openbsd_code/include/arpa/inet.h        inet_aton
4       e:\openbsd_code/include/arpa/inet.h        inet_lnaof
2       e:\openbsd_code/include/arpa/inet.h        inet_netof
5       e:\openbsd_code/include/arpa/inet.h        inet_network
625     e:\openbsd_code/include/arpa/inet.h        inet_ntoa
...
```

3.2.2.2 Steps 4 - 6

In order to be able to filter the functions based on support for Frama-C and its Jessie plug-in, a function's implementation is needed. As it would be expensive to search the code base one by one for each function, these steps follow a similar approach to the previous group of scripts and scan the code base once to find all source files that have function implementations in them and list the functions they implement.

There is also no need to scan all the functions implementations unnecessarily; so only the most popular functions are scanned as they are assumed to have higher utility for the purpose of the project. Therefore, the script at step 4 takes a threshold value and filters the output for the most popular functions to generate a smaller set of functions to work with.

Using the filtered list and the file generated by step 5, step 6 adds a column to the filtered text file for the source file where the function is implemented.

3.2.2.3 Step 7

This script goes through the text file generated by the previous step; opens each implementation file and looks for unsupported features like function pointers, unions and casts (see section 3.2.1). Finally, this step generates the final text file that includes an extra column for the following symbols:

- '!' for a function that requires further manual inspection. For example, a *goto* or a cast is found but this might not necessarily be a problem.

- '-' for a function that contains an unsupported feature such as a variadic function.

- '+' for a function that is a possible candidate.

- '?' for a function that requires a further manual check. This can be a case where an implementation file for a function is not found or multiple ones are found.

Although effort was made to make the scripts as accurate as possible, inevitably they did contain some bugs and therefore some manual triage was required in the end. However, this was acceptable for the purpose of the project as the intention was not to automate the whole triage but rather to inform and simplify it.

The following snippet shows an example of the final output of the process:

```
+  1476  /include/string.h  strcat   /lib/libc/string/strcat.c
-  19430 /include/stdio.h   fprintf  /lib/libc/stdio/fprintf.c
!  777   /include/string.h  strrchr  /sys/lib/libkern/strrchr.c
```

In the example above, the first column flags the outcome of the triage as explained earlier. The function on the first line is a possible candidate but a manual inspection is still required; for example for the complexity of the function for analysis. The second function is ruled out because it contains an unsupported construct, a variadic function in this case. The last line flags a warning for a construct that needs further manual triaging. A manual inspection reveals that it is a cast operation which is not an arbitrary cast and therefore is acceptable in this case.

3.2.2.4 Deciding on the functions

Following the triage process described in the previous section, the string functions were chosen for the examination. Choosing a set of related functions was preferred for the project and many string functions came up high in the list of most used functions. Even with a threshold value of 500 as a hit count, 11 of the 12 string functions eventually picked were in the list. Therefore, it was deemed appropriate that they had high utility and were critical for the OpenBSD code base.

In addition, the complexity of the implementation of most string functions met the complexity targeted for this project and they mostly did not use the unsupported features for the tools as discussed in the previous section.

A final manual triage was done in terms of complexity and unsupported features by working down the list of most popular string functions. The functions, that were flagged because they either had unsupported features or the triage was not conclusive, were manually inspected. Some functions were skipped as a result as they had either unsupported constructs for example, backward *goto*s or *(void *)* casts.

The following shows a snippet from the top of the candidate list:

```
10680  e:\openbsd_code/include/string.h      strcmp
9778   e:\openbsd_code/include/string.h      strlen
4662   e:\openbsd_code/include/string.h      memcpy
2892   e:\openbsd_code/include/string.h      strncmp
2739   e:\openbsd_code/include/string.h      strchr
2597   e:\openbsd_code/include/string.h      Strlcpy
. . .
```

The following is the final set of functions picked as candidates for examination in the order of utility: *strcmp*, *strlen*, *strncmp*, *strchr*, *strlcpy*, *strcpy*, *strcat*, *strrchr*, *strncpy*, *strstr*, *strlcat*, *strncat*.

3.3 Methodology for the contracts and the analysis

The goal of this project was to learn more about writing useful documentation, primarily in the form of contracts, for C program code in OpenBSD. Therefore, each of the string functions picked by triage process had several versions of specifications written. First one was based on the standard and the author's experience. The second one was written based on the manual page alone and the final one was based on the program code and the manual page. By comparing these specifications, it was attempted to determine if and where each informal specification was incorrect and to propose revisions to improve clarity and correctness.

The specifications for the string functions are discussed in detail in Chapter 4. The discussion will focus mainly on the final versions of the specifications; however other versions of the specifications and the differences will also be discussed as needed. All versions can be found on the project *Trac* site [25] [27].

The specifications were added to the string functions using ACSL at the source code level and these functions were analysed using the Frama-C Framework and its Jessie plug-in. The next section discusses checking the correctness of the functions with respect to the specifications.

In general, for writing the specifications, the advice from the paper "Specifications Tips and Pitfalls" was followed [28]. Using Frama-C and its Jessie plug-in, first the safety of the functions was proven and then it was attempted to verify the functional properties, like the input-output behaviour.

The specifications were added incrementally starting with simple specifications and adding more complex ones later as needed. The pre-conditions and the post-conditions were added first to limit the inputs and outputs of each method and this was followed by adding the loop invariants. In most cases, the loop invariants needed to be strengthened in order to prove the verification conditions generated.

The conjunctions were separated so that it was clear which conjunction was violated. This is also important for *requires* statements for the callers of these functions. *Assert* statements were also used to help track issues with the proof.

For more complex functions, the post-conditions were separated using behaviours. Using behaviours allowed being more precise about the corner cases and helped track issues while proving the verification conditions. The *assumes* clause of behaviours precisely gives the context in which behaviour applies. Behaviours for the success and failure cases were used to clarify the specifications.

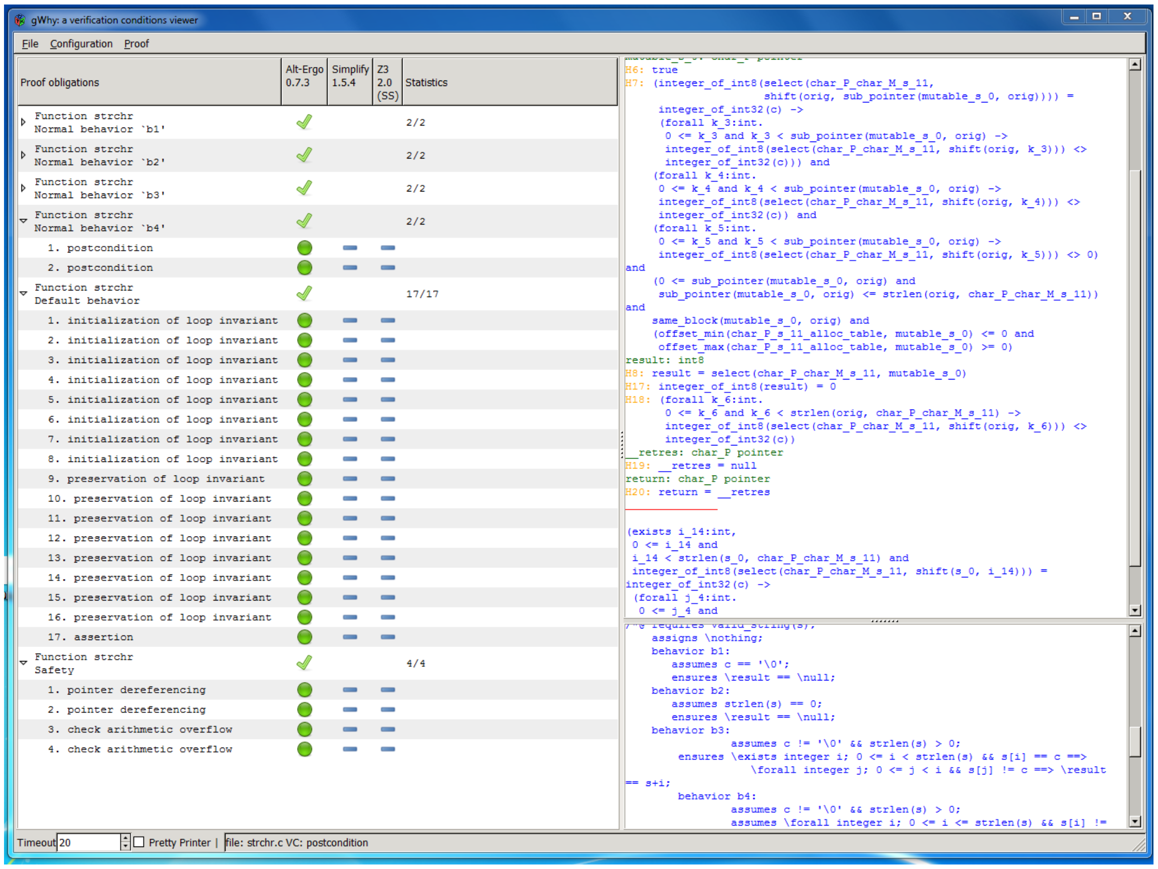
For all of the functions, the final version always started with the second version; the mistakes were found and fixed in the final version while proving the generated verification conditions using the tools.

3.3.1 Checking the correctness with respect to specifications

The string functions were analysed using the Frama-C Framework, the Jessie plug-in, the Why Platform and several theorem provers. As explained in Section 2.5.2, this meant that during the analysis, each C function was first translated to CIL, then to Jessie by the tools and finally the verification conditions or proof obligations were generated by the tools. These verification conditions were automatically sent to the theorem provers to prove the safety of the C functions and verify the functional properties by proving their correctness according to the specifications written.

Figure 3 shows the output for the *strchr* function as an example of proving proof obligations (or verification conditions) for checking the correctness of the code with respect to the specifications.

The verification was performed completely automatically and each function was analysed separately. A multi-prover approach, where the provers were run in parallel, was used for each function. All the generated proof obligations were proved using the Simplify, Alt-Ergo and Z3 theorem provers [26]. In some cases, one theorem prover was able to prove all proof obligations and in other cases a second theorem prover was used to prove the missing proof obligations. Overall, Alt-Ergo was able to prove the simpler functions on its own. Z3 and Simplify theorem provers performed better as the functions got more complex. Although, Simplify was by far the fastest to run and therefore it was usually the first theorem prover run to catch early issues.


gWhy: a verification conditions viewer
File Configuration Proof
Timeout 20
Pretty Printer
file: strchr.c VC: postcondition

Proof obligations	Alt-Ergo 0.7.3	Simplify 1.5.4	Z3 2.0 (SS)	Statistics
Function strchr Normal behavior `b1'	✓			2/2
Function strchr Normal behavior `b2'	✓			2/2
Function strchr Normal behavior `b3'	✓			2/2
Function strchr Normal behavior `b4'	✓			2/2
1. postcondition	●	–	–	
2. postcondition	●	–	–	
Function strchr Default behavior	✓			17/17
1. initialization of loop invariant	●	–	–	
2. initialization of loop invariant	●	–	–	
3. initialization of loop invariant	●	–	–	
4. initialization of loop invariant	●	–	–	
5. initialization of loop invariant	●	–	–	
6. initialization of loop invariant	●	–	–	
7. initialization of loop invariant	●	–	–	
8. initialization of loop invariant	●	–	–	
9. preservation of loop invariant	●	–	–	
10. preservation of loop invariant	●	–	–	
11. preservation of loop invariant	●	–	–	
12. preservation of loop invariant	●	–	–	
13. preservation of loop invariant	●	–	–	
14. preservation of loop invariant	●	–	–	
15. preservation of loop invariant	●	–	–	
16. preservation of loop invariant	●	–	–	
17. assertion	●	–	–	
Function strchr Safety	✓			4/4
1. pointer dereferencing	●	–	–	
2. pointer dereferencing	●	–	–	
3. check arithmetic overflow	●	–	–	
4. check arithmetic overflow	●	–	–	

```
H6: true
H7: (integer_of_int8(select(char_P_char_M_s_11,
                   shift(orig, sub_pointer(mutable_s_0, orig)))) =
     integer_of_int32(c) ->
     (forall k_3:int.
      0 <= k_3 and k_3 < sub_pointer(mutable_s_0, orig) ->
      integer_of_int8(select(char_P_char_M_s_11, shift(orig, k_3))) <>
      integer_of_int32(c))) and
    (forall k_4:int.
     0 <= k_4 and k_4 < sub_pointer(mutable_s_0, orig) ->
     integer_of_int8(select(char_P_char_M_s_11, shift(orig, k_4))) <>
     integer_of_int32(c)) and
    (forall k_5:int.
     0 <= k_5 and k_5 < sub_pointer(mutable_s_0, orig) ->
     integer_of_int8(select(char_P_char_M_s_11, shift(orig, k_5))) <> 0)
and
    (0 <= sub_pointer(mutable_s_0, orig) and
     sub_pointer(mutable_s_0, orig) <= strlen(orig, char_P_char_M_s_11))
and
    same_block(mutable_s_0, orig) and
    (offset_min(char_P_s_11_alloc_table, mutable_s_0) <= 0 and
     offset_max(char_P_s_11_alloc_table, mutable_s_0) >= 0)
result: int8
H8: result = select(char_P_char_M_s_11, mutable_s_0)
H17: integer_of_int8(result) = 0
H18: (forall k_6:int.
      0 <= k_6 and k_6 < strlen(orig, char_P_char_M_s_11) ->
      integer_of_int8(select(char_P_char_M_s_11, shift(orig, k_6))) <>
      integer_of_int32(c))
__retres: char_P pointer
H19: __retres = null
return: char_P pointer
H20: return = __retres

(exists i_14:int,
 0 <= i_14 and
 i_14 < strlen(s_0, char_P_char_M_s_11) and
 integer_of_int8(select(char_P_char_M_s_11, shift(s_0, i_14))) =
integer_of_int32(c) ->
 (forall j_4:int.
  0 <= j_4 and
```

```
    assigns \nothing;
    behavior b1:
       assumes c == '\0';
       ensures \result == \null;
    behavior b2:
       assumes strlen(s) == 0;
       ensures \result == \null;
    behavior b3:
                assumes c != '\0' && strlen(s) > 0;
        ensures \exists integer i; 0 <= i < strlen(s) && s[i] == c ==>
                   \forall integer j; 0 <= j < i && s[j] != c ==> \result
== s+i;
        behavior b4:
                assumes c != '\0' && strlen(s) > 0;
                assumes \forall integer i; 0 <= i <= strlen(s) && s[i] !=
```

Figure 3. *Proving proof obligations for strchr.*

As shown in Figure 3, all the proof obligations generated from the specifications were proven.

3.3.2 Jessie use

This section discusses some of the Jessie plug-in specific implementation details for the string functions.

The following is the command line used to prove specifications written:

```
frama-c -jessie -pp-annot -cpp-extra-args "-include spec_util.h" some_file.c
```

where:

- *-jessie* option activates the plug-in
- *-pp-annot* option makes symbols defined using *#define* recognized by Frama-C
- *-cpp-extra-args* option includes the header file for logic functions *strlen* and predicate *valid_string* (see section 4.2.2 for more information).

Note that the command line did not use *-jessie-no-regions* option so the default behaviour of memory region separation was used. By default, the Jessie plug-in assumes different pointers point into different memory regions.

3.3.3 The semantics of gWhy's icons

The user interface uses several different icons to indicate the result of the analysis of each proof obligation. The actual form and shape of the icons can be different depending on the GTK theme being used but the meaning of the icons is as follows [3]:

- Valid (a green dot): the verification condition is a true formula.
- Invalid (a red dot): the verification condition is not a true formula.
- Timeout (scissor): the verification condition was not proven within the specified timeout value.
- Failure (tools): the prover failed.
- Cached (hard disk): the verification condition has already been proven in a previous run of the tool and was cached.

In some cases, in order to prove the verification conditions of string functions, it helped to increase the default timeout value to 20s. Cached values were a great help to as it meant that not all the proof obligations had to be proved again in subsequent runs of the tools.

3.3.4 Verification condition groupings

As seen in Figure 3, generally the verification conditions are grouped in few groups. The proof of a verification condition in one group can be used to prove verification conditions in other groups. The verification conditions are grouped as follows [24]:

- Safety verification conditions that cover safety violations such as memory and integer safety.

- Default behaviour verification conditions that cover the default behaviour of a function such as post-conditions, frame conditions and loop invariants.

- User defined behaviour verification conditions that cover the user defined behaviours. The verification conditions in this group are verification conditions such as post-conditions, frame conditions and loop invariants that are specific to a behaviour.

4 Contracts for OpenBSD String Functions

This chapter discusses the contracts written for the twelve functions picked for examination as discussed in the previous chapter. Chosen based upon their size, complexity, and utility; these twelve string functions come from the string library in the *OpenBSD* kernel.

The following is the final set of functions picked as the first six functions for examination: *strcmp, strlen*, *strncmp*, *strchr*, *strcpy* and *strcat*.

The following six string functions constitute the second set chosen to be examined: *strrchr*, *strstr*, *strncpy*, *strncat*, *strlcpy* and *strlcat.*

Note that Section 3.3 discusses how the tools are used for checking the code with respects to the contracts and this chapter focuses on the contracts themselves.

4.1 String Functions Issues and Cleanup in OpenBSD

Motivated by the importance of security issues, in particular buffer overflow, different versions of string functions exist. For example, the string functions such as *strcpy* and *strcat* are deemed as dangerous functions and therefore length-bounded versions *strncpy* and *strncat* are the recommended safer versions. However, these length-bounded functions themselves are not free of issues either. This is mostly due to their misuse which has also been apparent from the auditing of the *OpenBSD* source tree for security issues. Therefore, new functions *strlcpy* and *strlcat* replace these functions. These new functions take the size of the string length in bytes, guarantee null termination and provide an easy way to detect truncation. *OpenBSD* project has been going through a cleanup effort to replace unsafe string functions with their safer versions such as *strlcpy* and *strlcat* [29].

4.2 Commonalities in the Contracts

This section is a high level discussion of the use of the basic annotation constructs for the contracts written and the commonalities among the contracts. The contracts for the twelve string functions presented in detail in the rest of this chapter share patterns and concepts to express the behaviour and the properties of the functions such as the pre-conditions, the post-conditions, the frame conditions and the invariants.

While each function contract is further presented and discussed in more detail individually in the following sections of this chapter, the commonalities in the contracts and the basics are discussed now in this section so that the same discussion is not repeated for each function contract and its discussion.

Section 2.3.1 introduces the basics of function contracts and the basic annotation constructs in ACSL. This section rather discusses the use of these annotations in the contracts written for the string functions.

Using actual examples and snippets from the contracts written, the following subsections discuss the common patterns among the contracts. In the following snippet, examples of these commonalities and use of built-in constructs are shown:

```
1 /*@
2   requires valid_string(s);
3   requires disjoint_strings(s, append);
4   requires \valid_range(s, 0, strlen(s) + strlen(append));
5   assigns s[..];
6   ensures \forall integer i; 0 <= i < strlen{Old}(s) ==> s[i] == \old(s[i]);
7   ensures \result == s;
8   . . .
9 */
10 char *
11 strcat(char *s, const char *append)
12 {
13   . . .
14   /*@ loop assigns s;
15       loop invariant \valid(s);
16       . . .
17   */
18   for (; *s; ++s);
19   . . .
20   //@ ghost char *s_cat = s;
21   /*@ loop invariant \forall integer k; (append-origAppend) <= k <= strlen(origAppend) ==>
22           origAppend[k] == \at(append[k], Pre);
23       . . .
24   */
25   while ((*s++ = *append++) != '\0');
26   //@ assert s_cat[append-origAppend - 1] == 0;
27   . . .
28   return(save);
29 }
```

Lines 2, 4 and 15 show how the contract checks the validity of strings, buffers and pointers as well as how the logic function *strlen* is used for referring to the length of strings. These are discussed in the first two subsections.

Line 3 shows how memory separation is ensured and this is discussed in Section 4.2.3.

Line 5 shows a frame condition for the function and this is discussed in Section 4.2.4 along with *loop assigns* clause at Line 14.

Lines 6 and 7 show how built-in constructs such as constructs for qualification and a function's return value are used. These built-in constructs are discussed in Section 4.2.5.

Although the example above uses only the default behaviour, many of the contracts written use named behaviours which are discussed in Section 4.2.6.

Line 15 expresses an invariant for a loop. Loop invariants are discussed in Section 4.2.7.

Lines 6 and 22 show how a function refers to values in pre-state and this is discussed in Section 4.2.8.

Line 20 shows a ghost variable use and this is discussed in Section 4.2.9.

Finally, Line 26 shows an example of assertions which are discussed in Section 4.2.10.

4.2.1 Checking validity of pointers and buffers

All the contracts written for the string functions require checking the validity of pointers and buffers.

In the following example, the specification states that the pointer p is valid and can be dereferenced safely:

```
\valid(p)
```

Similarly, in the following example, the pre-condition expresses that the buffer parameter *dst* is required to be valid from index 0 to index *siz*.

```
  requires \valid_range(dst, 0, siz);
```

4.2.2 Checking the validity of strings and finding the length of a string

All of the contracts written for the string functions have string parameters and checking that they are valid, null-terminated strings is a common task in all of the contracts. In addition, all of the contracts refer to the length of strings.

As mentioned in section 3.3.2, the logic function *strlen* and the predicate *valid_string* are included using a new header file and are used for the string specifications.

Strlen and v*alid_string* were included in the earlier Frama-C distribution originally but then were removed. Their implementations are discussed in the paper "Automatic Modular Static Safety Checking for C Programs" [15].

Valid_string predicate is simple and uses the *strlen* logic function:

```
  /*@ predicate valid_string{L}(char *s) =
    @   0 <= strlen(s) && \valid_range(s,0,strlen(s));
    @*/
```

For example, the function *strcat* requires that the strings are valid and null-terminated strings:

```
  /*@
    requires valid_string(s);
    requires valid_string(append);
    . . .
  */
  char *
  strcat(char *s, const char *append)
  {
```

In another pre-condition of *strcat*, the buffer is required to be big enough to hold the string *s* and *append*. Note that the actual length of the string is not the same as the buffer size:

```
  requires \valid_range(s, 0, strlen(s) + strlen(append));
```

Also, note the use of *strlen* logic function above.

4.2.3 Ensuring memory separation

At the time of the writing, the *\separated* annotation was not supported yet. As the Jessie plug-in works with a typed memory model by default, the following predicates were written to express the separation of buffers in the contracts for the string functions:

```
  /*@ predicate disjoint_strings{L}(char *s1, char *s2) =
       \forall integer i, j; 0 <= i < strlen(s1) && 0 <= j < strlen(s2) ==> s1 + i != s2 + j;
  */

  /*@ predicate disjoint_strings_len{L}(char *s1, char *s2, integer l) =
       \forall integer i, j; 0 <= i <= l && 0 <= j <= strlen(s2) ==> s1 + i != s2 + j;
  */
```

The *valid_string* predicate is not part of these predicate definitions because these predicates are only used internally and all the function specifications that use these predicates also use the *valid_string* predicate in the pre-conditions. Also note that despite its name the second predicate actually expects a buffer as first parameter not necessarily a string.

In the following example from the contract of the function *strcat*, the pre-condition states that the strings are required not to overlap:

```
/*@
  . . .
  requires disjoint_strings(s, append);
  requires disjoint_strings_len(s, append, strlen(s) + strlen(append));
  . . .
*/
char *
strcat(char *s, const char *append)
{
```

If the second string happens to be located after the first string in memory, then the second string can get overwritten. Therefore, the second pre-condition requires that the part of the buffer, where the characters from the second string will be copied to, should not overlap with the second string either.

4.2.4 Assignments

When a string function such as *strchr* does not modify anything, the following annotation is used to express this:

```
assigns \nothing;
```

When a function such as *strcat* writes to a buffer, the following annotation is used instead:

```
assigns s[..];
```

The annotation above does not specify a range and could be made more explicit. For example, the following is an example that uses a more specific range:

```
assigns s[0..siz-1];
```

However, specifying a specific range is not always possible for the functions examined with the versions of the tools used for the project. This is further discussed in the sections that discuss the individual functions and their contracts.

In addition to the frame conditions, the loops can also assign to variables. The loop assignments are expressed using the *loop assigns* annotations and help prove the frame conditions of a function.

In the example below, an annotated loop in the function *strlcat* is shown. The variable *n*, the pointers *d* and *s* are modified (incremented or decremented) within the loop and the loop also writes to the contents of the buffer *dst*:

```
char *d = dst;
/*@ loop assigns n, s, d, dst[p-dst..];
    . . .
*/
while (*s != '\0') {
  if (n != 1) {
    *d++ = *s;
    n--;
  }
  s++;
}
```

4.2.5 Using the built-in constructs for quantification and function results

The result of a function is denoted by the built-in construct *\result*. In the following example, one of the possible return values of the function *strchr* is specified to be null which is denoted by the built-in construct *\null*:

```
ensures \result == \null;
```

In ACSL, universal quantification is denoted by *\forall*. In the following example, one of the post-conditions of the function *strcmp* expresses that the result of the function will be zero if the strings are equal:

```
ensures \forall integer i; 0 <= i <= strlen(s1) && s1[i] == s2[i]) ==> \result == 0;
```

In the example above, the universal quantification is used for the comparison of all the characters found at the indices from zero to the length of string *s1* including the terminating null-character.

The existential quantification is denoted by the built-in construct *\exists* in ACSL. In the following example, one of the post-conditions of the function *strcmp* is shown. In this post-condition, it is expressed that the function returns a value that is less than zero if there exists an index value at which the value found in the first string is smaller than the one found at the same index in the second string:

```
ensures \exists integer i; 0 <= i <= strlen(s1) && 0 <= i <= strlen(s2) && s1[i] <  s2[i]  ==> \result < 0;
```

4.2.6 Using named behaviours

Most of the contracts written for the string functions use named behaviours. Using behaviours simplifies writing the specifications, improves readability and also simplifies proving some of the more complex specifications as they are written in simpler specifications that apply only to specific behaviours. In the following example, the function *strncpy* uses named behaviours to express when the function modifies the buffer *dst*:

```
/*@
  . . .
  ensures \result == dst;
  behavior b1:
    assumes n == 0;
    assigns \nothing;
    . . .
  behavior b2:
    assumes n > 0 && strlen(src) > n;
    assigns dst[..];
    . . .
  behavior b3:
    assumes n > 0  && strlen(src) < n;
    assigns dst[..];
    . . .
*/
```

In the example above, *strncpy* modifies nothing if *n* is zero as expressed in behaviour *b1*. If *n* is not zero, then the function writes to the buffer *dst* as expressed in the named behaviours. Furthermore, the function behaves differently depending on whether the length of the string is less than or greater than *n*. The result of the function as denoted by the built-in construct *\result* is expressed in the default behaviour and it applies to all the behaviours.

The loop invariants, which are discussed in the next section, can also be expressed to apply to certain behaviours only. In the following example from the contracts for the function *strrchr*, the loop invariant is specific to behaviour *b1* and states that the pointer *t* is always *null* for that behaviour:

```
for b1: loop invariant t == \null;
```

4.2.7 Using loop invariants for annotating loops

Almost all string functions contain loops; therefore loop annotations using *loop invariant* are common to most of the function contracts. The loop invariants help proving the post-conditions of a function.

In the example below, a *while* loop in the function *strncat* is annotated using *loop invariant* annotations:

```
char *d = dst;
/*
  . . .
  loop invariant \base_addr(d) == \base_addr(dst);
  loop invariant \valid(d);
  loop invariant 0 <= (d - dst) <= strlen(dst);
  loop invariant \forall integer i; 0 <= i < (d-dst) ==> dst[i] != 0;
*/
while (*d != 0)
  d++;
```

The loop invariants above express that the pointer *d*, which is initially equal to pointer *dst*, is incremented during the loop but not beyond the end of the string. Therefore, the pointers *d* and *dst* have the same base address and pointer *d* remains as a valid pointer at each iteration of the loop. As the pointer *d* is valid during the loop, it is therefore safe to dereference it during the loop. Finally, at any point during the loop iteration, all the characters visited up to that point are not zero.

4.2.8 Referring to the values in the pre-state of a function

Referring to the old values of parameters prior to the function call is essential in most of the contracts for the string functions. Section 2.3.1.4 explains the built-in constructs and the logic labels in ACSL.

The logic function *strlen* is used in almost all of the contracts written. As it is a logic function, *strlen* can be passed a label for state and this can be used to refer to the pre-state. For example, the function *strcat* refers to the original length of string *s* as follows:

```
strlen{Old}(s)
```

The following example shows one of the post-conditions of the function *strcat* where the original length of string *s* is referred to:

```
ensures strlen(s) == strlen{Old}(s) + strlen(append);
```

The post-condition above expresses that the length of the resulting string *s* is the original length of *s* plus the length of the string *append*.

Another post-condition of the function *strcat* is shown below where built-in construct *\old* is used to refer to the pre-state. This post-condition expresses that the characters up to the original length of the string are not modified and are the same as the previous values prior to the function call:

```
ensures \forall integer i; 0 <= i < strlen{Old}(s) ==> s[i] == \old(s[i]);
```

Another logic label that can be seen in the contracts written for the string functions is *Pre* which must be used in loop invariants to refer to the values prior to the function entry. The following example for using *Pre* is from the contract of the function *strncat*:

```
loop invariant \forall integer i; (d-dst) <= i <= strlen{Pre}(dst) ==> dst[i] == \at(dst[i], Pre);
```

The construct *\at* is a more general form where the state label can be passed as the second parameter. The construct *\old* is only a shorter version of *\at(.., Old)*. Therefore the example above uses *\at* construct rather than *\old* since logic label *Pre* is used in loops as explained above.

4.2.9 Using ghost variables

As explained in Section 2.3.1, ACSL allows using ghost code and ghost variables in the specifications. For the contracts written for the string functions, the use of ghost code has been avoided but in some cases ghost variables are used to hold temporary values to simplify the specifications. Although sometimes ghost variables can be avoided, for example using the labels explained in previous section, sometimes it is necessary to use ghost variables to store intermediate values. In addition, using ghost variables in some cases simply shortens the specifications and improves readability.

In the following example, the specifications for the function *strncpy* use the ghost variable *n2* to store the value of variable *n* before the inner loop is entered:

```
do {
  if ((*d++ = *s++) == 0){
    /* NUL pad the remaining n-1 bytes */
    . . .
    //@ ghost size_t n2 = n;
    /*@ loop assigns d, n, dst[p-dst..];
        . . .
        loop invariant \at(n, Pre) >= n2 >= n > 0;
        . . .
    */
    while (--n != 0)
      *d++ = 0;
    break;
  }
}
while (--n != 0);
```

In the next example, a ghost variable is used in the specifications for the function *strcat* to store the pointer value before the start of the loop:

```
//@ ghost char *s_cat = s;
. . .
/*@ loop assigns s, save[s_cat-save..], append;
    . . .
    loop invariant (append-origAppend) == (s - s_cat);
    . . .
```

This ghost variable is used above in order to be able to express the range of the buffer that is modified:

```
save[s_cat-save..]
```

The ghost variable in this case can be avoided using *strlen{Pre}(save)* instead; however the version of the tools used did not support labels in the loop assigns and this is why the ghost variable is used. It is also less verbose to express other loop invariants based on this temporary value:

```
loop invariant (append-origAppend) == (s - s_cat);
```

4.2.10 Use of assertions

Most of the contracts written for the string functions contain assertions that help proving the specifications. The assertions help understanding why some proof obligations cannot be proven and are necessary in some cases to provide clues to the theorem provers in order to prove the specifications.

When a proof obligation generated for the specifications of a function is not proven by the theorem provers, it helps to insert assertions in the code to express intermediate properties of the code. The assertions added also generate proof obligations and proving those can be important to understand why a proof obligation, for example for a post-condition, is not proven. The assertions are also used by the theorem provers in the proof of the other proof obligations. Therefore using assertions is a way to provide clues or hints to the theorem provers to use in the proof of the specifications.

In the example below, an assertion added for the function *strlcat* is shown:

```
size_t n = siz;
. . .
while (n-- != 0 && *d != '\0')
  d++;
dlen = d - dst;
//@ assert dlen <= siz;
n = siz - dlen;
if (n == 0)
. . .
```

The assertion above is used to help proving the safety proof obligation generated for the subtraction of *dlen* from *siz* by asserting and hinting that the value of *dlen* is not greater than *siz*. This information can be used to prove that this subtraction operation is actually safe and will not result in a negative value which could subsequently cause an overflow as the variable *n* is an unsigned integer.

4.3 Contracts

This section discusses each of the function contracts written for the *OpenBSD* string functions. The discussion on commonalities in the contracts and the use of basic constructs can be found in Section 4.2. Note that all the source code discussed here belong to the *OpenBSD project* [12]. Each subsection for a function starts with a short description of the function based on its man page.

4.3.1 Strchr

The *strchr* function locates the first occurrence of the character *c* in the string *s*. The terminating *null* character is considered to be part of the string. If *c* is '\0', then *strchr* locates the terminating null character. The *strchr* function returns a pointer to the located character or *null* if the character does not appear in the string.

4.3.1.1 Contracts

First version of the contract was as follows:

```
/*@ requires valid_string(s);
    assigns \nothing;
    ensures \exists integer i; 0 <= i < strlen(s) && s[i] == c ==>
      \result == s+i;
    ensures \forall integer i; 0 <= i < strlen(s) && s[i] != c ==>
      \result == \null;
*/
char *strchr(const char *s, int c);
```

The manual version of the contracts was as follows:

```
/*@ requires valid_string(s);
    assigns \nothing;
    ensures \exists integer i; 0 <= i <= strlen(s) && s[i] == c &&
       \forall integer j; 0 <= j < i && s[j] != c ==> \result == s+i;
    ensures \forall integer i; 0 <= i <= strlen(s) && s[i] != c ==>
       \result == \null;
*/
char *strchr(const char *s, int c);
```

The main difference between the two contract versions is that the return pointer points to the first occurrence of the character. The man version corrects this in its first post-condition.

The code and the final contracts for *strchr* are given in Listing 1.

```
 1 /*@ requires valid_string(s);
 2     assigns \nothing;
 3     behavior b1:
 4       assumes c == '\0';
 5       ensures \result == \null;
 6     behavior b2:
 7       assumes strlen(s) == 0;
 8       ensures \result == \null;
 9     behavior b3:
10       assumes c != '\0' && strlen(s) > 0;
11       ensures \exists integer i; 0 <= i < strlen(s) && s[i] == c ==>
12         \forall integer j; 0 <= j < i && s[j] != c ==> \result == s+i;
13     behavior b4:
14       assumes c != '\0' && strlen(s) > 0;
15       assumes \forall integer i; 0 <= i <= strlen(s) && s[i] != c;
16       ensures \result == \null;
17  */
18 char *
19 strchr(const char *s, int c)
20 {
21   //@ ghost char *orig = s;
22   /*@ loop assigns s;
23       loop invariant \valid(s);
24       loop invariant \base_addr(s) == \base_addr(orig);
25       loop invariant 0 <= (s-orig) <= strlen(orig);
26       loop invariant \forall integer k; 0 <= k < (s-orig) ==> orig[k] != 0;
27       loop invariant \forall integer k; 0 <= k < (s-orig) ==> orig[k] != c;
28       loop invariant orig[(s-orig)] == c ==> (\forall integer k; 0 <= k < (s-orig) ==> orig[k] != c);
29   */
30   while (*s) {
31     if (*s == c)
32       return ((char *)s);
33     s++;
34   }
35   //@ assert \forall integer k; 0 <= k < strlen(orig) ==> orig[k] != c;
36   return (NULL);
37 }
```

Listing 1. *Strchr function and its contracts.*

The inspection of the final version and the version based on manual reveals that there is a manual bug:

```
<<<ensures \exists integer i; 0 <= i <= strlen(s) && s[i] == c &&
       \forall integer j; 0 <= j < i && s[j] != c ==> \result == s+i;
>>>ensures \exists integer i; 0 <= i < strlen(s) && s[i] == c ==>
       \forall integer j; 0 <= j < i && s[j] != c ==> \result == s+i;
```

The equal sign was removed so the post-condition could be proven. Although the manual states the following:

```
“The terminating NUL character is considered part of the string. If c is ‘\0’, strchr() locates the
terminating ‘\0’.”
```

The code states otherwise; when *c* is '\0', the function returns *null*. So, the behaviour *b1* at line 3 in the final version was added to cover this.

This also means that if the string's length is zero, then the function also returns *null* and behaviour *b2* at line 6 expresses this. Note that, according to the man page, this function would not always return *null* for an empty string.

The final version uses behaviours because it makes proving the proof obligations and narrowing down any issues easier.

4.3.1.2 Proof Results

All proof obligations are proven by Alt-Ergo:

```
b1: 2/2, b2: 2/2, b3: 2/2, b4: 2/2, default behaviour 17/17, safety: 4/4.
```

4.3.1.3 Specification Details

The function requires that the string is valid and null-terminated. The function only reads the buffer and does not modify it.

The post-condition at line 11 ensures that if the function's return value is not *null* then it is a pointer to the character *c* and any character before this location is not *c*; therefore it's the first occurrence.

The post-condition at line 16 states that the function returns *null* if the character *c* is not found in the string.

Using the assertion at line 35, it is verified that the character is not found in the string if the function is returning *null*.

4.3.2 Strcmp

The *strcmp* function lexicographically compares the null-terminated strings *s1* and *s2*. The *strcmp* returns an integer greater than, equal to, or less than zero, according to whether the string *s1* is greater than, equal to, or less than the string *s2*. The comparison is done using unsigned characters, so that '\200' is greater than '\0'.

4.3.2.1 Contracts

First version of the contracts is as follows:

```
/*@ requires valid_string(s1) && valid_string(s2);
    assigns \nothing;
    behavior same_strings:
      assumes strlen(s1) == strlen(s2) &&
        \forall integer i; 0 <= i < strlen(s1) && s1[i] == s2[i];
      ensures \result == 0;
    behavior s1_smaller:
      assumes strlen(s1) < strlen(s2) || (strlen(s1) == strlen(s2)
        && \exists integer i; 0<=i<strlen(s1) && s1[i] < s2[i]);
      ensures \result == -1;
    behavior s2_smaller:
      assumes strlen(s2) < strlen(s1) || (strlen(s1) == strlen(s2)
        && \exists integer i; 0<=i<strlen(s2) && s2[i] < s1[i]);
      ensures \result == 1;
*/
int strcmp(const char *s1, const char *s2);
```

Man version of the contracts is as follows:

```
/*@ requires valid_string(s1) && valid_string(s2);
    assigns \nothing;
    behavior same_strings:
      assumes strlen(s1) == strlen(s2) &&
          \forall integer i; 0 <= i < strlen(s1) && s1[i] == s2[i];
      ensures \result == 0;
    behavior s1_smaller:
      assumes strlen(s1) < strlen(s2) || (strlen(s1) == strlen(s2) &&
        \exists integer i; 0<=i<strlen(s1) &&
          (unsigned char)s1[i] < (unsigned char) s2[i]);
      ensures \result < 0;
    behavior s2_smaller:
      assumes strlen(s2) < strlen(s1) || (strlen(s1) == strlen(s2) &&
        \exists integer i; 0<=i<strlen(s2) && (unsigned char) s2[i] <
          (unsigned char)s1[i]);
      ensures \result > 0;
*/
int strcmp(const char *s1, const char *s2);
```

First version differed from the second version in the return values as the first version returned 1 and -1; while the second version correctly returns a value less than or greater than zero.

The code and the final contracts are given in Listing 2.

```
/*@ requires valid_string(s1);
    requires valid_string(s2);
    assigns \nothing;
    ensures (strlen(s1) == strlen(s2) && \forall integer i; 0 <= i <= strlen(s1) &&
      s1[i] == s2[i]) ==> \result == 0;
    ensures \exists integer i; 0<=i<= strlen(s1) && 0<=i<= strlen(s2) && s1[i] <  s2[i]  &&
      (\forall integer k; 0 <= k < i ==> s1[k] == s2[k])  ==> \result < 0;
    ensures \exists integer i; 0<=i<= strlen(s1) && 0<=i<= strlen(s2) && s2[i] >  s1[i] &&
      (\forall integer k; 0 <= k < i ==> s1[k] == s2[k]) ==> \result > 0;
*/
int
strcmp(const char *s1, const char *s2)
{
  //@ ghost char *orig1 = s1;
  //@ ghost char *orig2 = s2;
  /*@ loop assigns s1, s2;
      loop invariant valid_string(orig1);
      loop invariant valid_string(orig2);
      loop invariant \valid(s1) && \valid(s2);
      loop invariant \base_addr(s1) == \base_addr(orig1);
      loop invariant \base_addr(s2) == \base_addr(orig2);
      loop invariant s1 - orig1 == s2 - orig2;
      loop invariant 0 <= (s2-orig2) <= strlen(orig2);
      loop invariant 0 <= (s1-orig1) <= strlen(orig1);
      loop invariant \forall integer k; 0 <= k < (s2-orig2) ==> orig1[k] != 0;
      loop invariant \forall integer k; 0 <= k < (s2-orig2) ==> orig1[k] == orig2[k];
  */
  while (*s1 == *s2++)
    if (*s1++ == 0)
      return (0);
  //!!!! return (*(unsigned char *)s1 - *(unsigned char *)--s2);
  return (*s1 - *(--s2)); //entered bug 306
}
```

Listing 2. *Strcmp function and its contracts.*

The final specification written matches the description from the manual except the cast to unsigned char. As noted in the section 3.2.1, pointer casts were problematic with the versions of the tools used. More specifically, the following statement is not supported:

```
return (*(unsigned char *)s1 - *(unsigned char *)--s2);
```

Therefore, the casts were removed as follows:

```
return (*s1 - *(--s2));
```

A bug (Bug #306) was entered in Frama-C bug database [3] and this issue was subsequently fixed in Why version 2.2. As the infrastructural changes were too late to make in the project, the fix was not verified.

The contracts version based on man file uses behaviors but final version uses conditional post-conditions.

While trying to prove the post-conditions, a bug was noticed in the second version of the specifications:

```
<<<
  behavior s1_smaller:
    assumes strlen(s1) < strlen(s2) || (strlen(s1) == strlen(s2) && \exists integer i; 0<=i<strlen(s1) &&
      (unsigned char)s1[i] < (unsigned char) s2[i]);
    ensures \result < 0;
>>>
  ensures \exists integer i; 0<=i<= strlen(s1) && 0<=i<= strlen(s2) && s1[i] <  s2[i]  &&
    (\forall integer k; 0 <= k < i ==> s1[k] == s2[k]) ==> \result < 0;
```

Final version correctly adds the part in bold font to express that the function returns less than zero if the "first" character that does not match is less than the respective character in the second string. The same issue was fixed in the behavior *s2_smaller*.

Also, the final version does not check string lengths anymore as an index that exists in both strings is used for comparison.

4.3.2.2 Proof Results

All proof obligations are proven by Alt-Ergo:

```
Default behavior: 36/36, safety: 20/20
```

4.3.2.3 Specification Details

The function requires that the strings are valid, null-terminated strings as a pre-condition and the function does not modify any of the buffers.

The post-condition at line 4 states that if the strings are the same length and all respective characters are the same then the function returns zero. The first part of the statement is used as a shortcut and also simplifies the second part of the expression as just the length of one of the strings can be used.

The post-condition at line 6 states that the function returns a value less than zero if there is one index that is:

- Smaller than lengths of both strings.
- The comparison of the values found in the strings at that index show that the character found in the first string is less than the one found in the second string and up to that character; all characters are the same if there are any.

Similarly, the post-condition at line 8 states that the function returns a value that is greater than zero if that character is greater than the respective character in the second string.

The loop invariants state that:

- The modified *s1* and *s2* pointers remain as valid strings during the loop iteration since the pointers are never incremented beyond the lengths of the original strings.
- At any stage in the loop iteration, the elements compared so far are identical.

4.3.3 Strncmp

Strncmp function lexicographically compares the null-terminated strings *s1* and *s2*. *Strncmp* returns an integer value greater than, equal to, or less than zero, according to whether the string *s1* is greater than, equal to, or less than the string *s2*. The comparison is done using unsigned characters, so that '\200' is greater than '\0'. *Strncmp* compares at most *len* characters.

4.3.3.1 Contracts

First version of contracts was as follows:

```
/*@ requires valid_string(s1) && valid_string(s2);
    requires len <= strlen(s1) && len <= strlen(s2);
    assigns \nothing;
    behavior same_strings:
      assumes \forall integer i; 0 <= i < len  && s1[i] == s2[i];
      ensures \result == 0;
    behavior s1_smaller:
      assumes \exists integer i; 0 <=i< len && s1[i] < s2[i];
      ensures \result == -1;
    behavior s2_smaller:
      assumes \exists integer i; 0<=i< len && s2[i] < s1[i];
      ensures \result == 1;
*/
int strncmp(const char *s1, const char *s2, size_t len);
```

The version based on the manual file was as follows:

```
/*@ requires valid_string(s1) && valid_string(s2);
    assigns \nothing;
    behavior same_strings:
      assumes \forall integer i; 0 <= i < minimum(len, minimum(strlen(s1), strlen(s2))) && s1[i] == s2[i];
      ensures \result == 0;
    behavior s1_smaller:
      assumes \exists integer i; 0<= i < minimum(len, minimum(strlen(s1), strlen(s2))) &&
        (unsigned char)s1[i] < (unsigned char) s2[i];
      ensures \result < 0;
    behavior s2_smaller:
      assumes \exists integer i; 0<= i< minimum(len, minimum(strlen(s1), strlen(s2))) &&
        (unsigned char) s2[i] < (unsigned char)s1[i];
      ensures \result > 0;
*/
int strncmp(const char *s1, const char *s2, size_t len);
```

First version differed from the second version in the return values as the first version returned 1 and -1; while the second version correctly returns a value greater than or less than zero.

The code and the final contracts are given in Listing 3.

Similar to the previous function *strcmp*, the final specification written matches the description from the manual except for the cast to unsigned char. As noted in the section 3.2.1, pointer casts are problematic so the tools used did not support the following statement:

```
return (*(unsigned char *)s1 - *(unsigned char *)--s2);
```

Therefore the casts were removed as follows:

```
return (*s1 - *(--s2));
```

A bug (Bug# 306) was entered in Frama-C bug database [3] and the issue was fixed in Why version 2.2 [13]. As the infrastructural changes were too late to make in the project, the fix was not verified.

4.3.3.2 Man Version versus Final Version

Parameter name *n* does not match the man file which is *len*, it was renamed in the final specification.

```
 1 /*@ requires valid_string(s1);
 2    requires valid_string(s2);
 3    requires n < INT_MAX;
 4    assigns \nothing;
 5    ensures n == 0 ==> \result == 0;
 6    ensures (n>0 ==> \forall integer i; 0<= i <= minimum(n-1, minimum(strlen(s1), strlen(s2)))
 7      && s1[i] == s2[i]) ==> \result == 0;
 8    ensures \result < 0 ==> (n > 0 && \exists integer i;
 9      0<= i <= minimum(n-1, minimum(strlen(s1), strlen(s2))) && s1[i] < s2[i] ==>
10        (\forall integer k; 0 <= k < i ==> s1[k] == s2[k])) ;
11    ensures \result > 0 ==> (n > 0 && \exists integer i;
12      0 <= i <= minimum(n-1, minimum(strlen(s1), strlen(s2))) && s1[i] > s2[i] ==>
13        (\forall integer k; 0 <= k < i ==> s1[k] == s2[k]));
14  */
15  int
16  strncmp(const char *s1, const char *s2, size_t n)
17  {
18    if (n == 0)
19      return (0);
20    //@ ghost char *orig1 = s1;
21    //@ ghost char *orig2 = s2;
22    //@ ghost int origN = n;
23    /*@ loop assigns s1, s2, n;
24        loop variant n;
25        loop invariant valid_string(orig1);
26        loop invariant valid_string(orig2);
27        loop invariant \valid(s1) && \valid(s2);
28        loop invariant \base_addr(s1) == \base_addr(orig1);
29        loop invariant \base_addr(s2) == \base_addr(orig2);
30        loop invariant s1 - orig1 == s2 - orig2;
31        loop invariant 0 <= (s2-orig2) <= strlen(orig2);
32        loop invariant 0 <= (s1-orig1) <= strlen(orig1);
33        loop invariant \forall integer k; 0 <= k < (s1-orig1) ==> orig1[k] != 0;
34        loop invariant \forall integer k; 0 <= k < (s1-orig1)==> orig1[k] == orig2[k];
35        loop invariant 0 < n <= origN;
36    */
37    do {
38      if (*s1 != *s2++)
39        return (*s1 - *(--s2)); //bug 306: return (*(unsigned char*)s1-*(unsigned char*)--s2);
40      if (*s1++ == 0)
41        break; //@assert *(s1-1) == *(s2-1);
42    } while (--n != 0);
43    return (0);
44  }
```

Listing 3. *Strncmp function and its contracts.*

The post-condition at line 5 was not included in the second version as the man file did not mention it. However, the man file needs to state that if *n* equals to zero then the return value is zero.

However, this behaviour does not seem to be the best design choice as the return value is mixing valid data with an error condition. This can go undetected in the client code as the caller might not realise that the strings are not actually equal but the parameter *n* is, possibly unintentionally, zero. Therefore, adding a pre-condition to this function for requiring *n* to be greater than zero would be a better solution.

The second version uses behaviours but the final version uses conditional post-conditions which is a style difference.

The following mistake was also fixed in the final version as the verification conditions were not proven otherwise:

```
<<< \forall integer i; 0 <= i < minimum(len, minimum(strlen(s1), strlen(s2)))
>>> \forall integer i; 0 <= i <= minimum(n-1, minimum(strlen(s1), strlen(s2)))
```

This is because of the zero based index. Note that there is no change in the second part:

```
minimum(strlen(s1), strlen(s2))
```

Similarly to *strcmp*, the following mistake was also corrected in the final version:

```
<<<
  behavior s1_smaller:
    assumes \exists integer i; 0<= i < minimum(len, minimum(strlen(s1), strlen(s2)))
      && (unsigned char)s1[i] < (unsigned char) s2[i];
    ensures \result < 0;
>>>
  ensures \result < 0 ==> (n > 0 && \exists integer i;
    0 <= i <= minimum(n-1, minimum(strlen(s1), strlen(s2))) && s1[i] < s2[i] ==>
      (\forall integer k; 0 <= k < i ==> s1[k] == s2[k])) ;
```

Final version correctly adds the part shown in bold font above as explained in previous section for *strcmp*.

Also, the final version does not check string lengths anymore as an index that exists in both strings is used for comparison.

4.3.3.3 Proof Results

All proof obligations are proven by Z3 and Alt-Ergo. Z3 proves all proof obligations except one:

```
Normal behavior: 58/59, safety: 26/26.
```

Alt-ergo proves the missing proof obligation which is for preservation of a loop invariant.

4.3.3.4 Specification Details

The pre-conditions state that the function requires the strings are null-terminated, valid strings. The function does not modify the buffers.

Post-condition at line 5 states that result will be zero if the count parameter is zero.

The post condition at line 6 states that the result will be also zero if the strings are equal. Only at most first *n* characters are compared. If the strings have different lengths, a shortcut is used by comparing only up to the length of the shorter string where two strings will differ.

The post condition at line 8 states that after the comparison, the result is less than zero if the comparison finds a character that is less than the respective char in the second string and up to this character all the characters are the same if there any. The comparison is done up to the shorter string and only done up to at most first n characters. Similarly the last post-condition states that the function returns a value that is greater than zero if that character is greater than the respective character.

The loop invariant at line 35 states that the count *n* starts with original value and is never decremented past zero. The loop invariant at line 34 states that at any stage in the loop, the elements compared so far are identical. There is no need to check for the null character in the second string as the comparison would fail.

The assertion at line 41 verifies that if the loop breaks there then it is the end of both strings that is two strings are identical.

4.3.4 Strlen

The *strlen* function computes the length of the string *s*. The *strlen* function returns the number of characters that precede the terminating null character.

4.3.4.1 Contracts

First version of contracts was as follows:

```
/*@ requires valid_string(s);
  @ assigns \nothing;
  @ ensures \result == strlen(s) && \forall unsigned int k; 0 <= k < \result && s[k] != '\0';
  @*/
size_t strlen(const char *s);
```

The second version based on man file was as follows:

```
/*@ requires valid_string(s);
  @ assigns \nothing;
  @ ensures \valid_range(s, 0, \result) && s[\result] == '\0' &&
  @   \forall unsigned int k; 0 <= k < \result && s[k] != '\0';
  @*/
size_t strlen(const char *s);
```

The code and the final contracts are given in Listing 4.

The first version is almost same as the final version but is unnecessarily longer.

```
/*@ requires valid_string(str);
    assigns \nothing;
    ensures \result == strlen(str);
*/
size_t
strlen(const char *str)
{
  const char *s;
  /*@
    loop invariant valid_string(s);
    loop invariant \base_addr(s) == \base_addr(str);
    loop invariant \forall integer k; 0 <= k < (s-str) ==> str[k] != '\0';
    loop invariant 0 <= (s-str) <= strlen(str);
  */
  for (s = str; *s; ++s)
    ;
  //@ assert *s == '\0';
  //@ assert str[s-str] == '\0';
  return (s - str);
}
```

Listing 4. *Strlen function and its contracts.*

4.3.4.2 Man Version versus Final Version

First thing to notice between the second and the final version is that the parameter names do not match the man file.

Valid range check is unnecessary as that is covered by the pre-condition that the string is valid null-terminated string.

The use of conjunction (*&&)* in the post-condition was also wrong and the tool failed to prove the case of empty string with length zero.

The result of the function is what the *Strlen* logic function returns which is the index of the first occurrence of null character. However, the man file does not mention that it is the first occurrence; for example some strings can be double-terminated. Although this may not be as important to mention in the man file, it constitutes an important part of the specifications.

Finally, the post-condition was replaced with the logic function as in the first version.

4.3.4.3 Proof Results

All proof obligations were proven by Alt-Ergo:

```
Default behavior: 13/13, safety: 5/5
```

4.3.4.4 Specification Details

The function requires that the string is a null-terminated, valid string. For example, if the string was not null terminated, then the function would not terminate.

The loop modifies the local pointer *s* and the loop invariants state that the pointer *s* is incremented during the loop and at any stage during the iteration the characters visited are not '\0'.

The assertions at lines 17 and 18 make it explicit that pointer *s* points at the null character in the array and *(s-str)*, which is the return value, is the number of the characters in the string up to the null character.

4.3.5 Strcpy

The *strcpy* function copies the string *src* to string *dst* including the terminating '\0' character. The *strcpy* function returns *dst*.

4.3.5.1 Contracts

First version of the contracts was as follows:

```
/*@ requires valid_string(s1) && valid_string(s2);
    assigns s1;
    behavior normal:
       assumes strlen(s1) >= strlen(s2);
       ensures \forall integer i; 0 <= i < strlen(s2) && s1[i] == s2[i] && \result == s1;
    behavior error:
       assumes strlen(s1) < strlen(s2);
       assigns \nothing;
       ensures \result == \null;
*/
char *strcpy(char *s1, const char *s2);
```

The version based on man file was as follows:

```
/*@ requires valid_string(dst) && valid_string(src);
    assigns dst[0..minimum(strlen(dst), strlen(src))];
    ensures \forall integer i; 0 <= i <= minimum(strlen(dst), strlen(src)) && dst[i] == src[i];
    ensures \result == dst;
*/
char *strcpy(char *dst, const char *src);
```

The first version assumes that the function returns *null* and assigns nothing for error condition where the destination string is shorter but this does not match the man page. If one considers a function such as the *strchr*, the string functions do not really behave consistently and this makes them hard to learn.

The following mistake was also corrected in the man version as the function changes the contents of the buffer, not the buffer:

```
>>> assigns s1;
```

```
<<< assigns dst[0..minimum(strlen(dst), strlen(src))];
```

The code and the final contracts are given in Listing 5.

4.3.5.2 Man Version versus Final Version

First thing to notice is that the parameter names do not match the man file.

The destination string was considered to be a valid, null-terminated string in the second version. However, inspecting the code along with the man file reveals that actually the destination string does not need to be null-terminated as the function simply writes over the buffer with no constraint apart from source string's length. Therefore, the final spec was changed to state that the destination pointer needs to be valid and the source string needs to be a null-terminated, valid string. For the same reason, parts of the specification had to be re-written to take out references to length of the destination string.

Additionally, memory separation pre-condition was added as the function behaviour is not safe if the buffers overlap.

Also, the following change had to be made as logic functions could not be used in *assigns* clauses with the version of the tools used:

```
<<< assigns dst[0..minimum(strlen(dst), strlen(src))];
>>> assigns to[0..];
```

Finally, the pre-condition at line 4 was added to protect against buffer overrun.

```
1   /*@ requires valid_string(from);
2       requires disjoint_strings(to, from);
3       requires disjoint_strings_len(to, from, strlen(from));
4       requires \valid_range(to, 0, strlen(from));
5       assigns to[0..];
6       ensures \forall integer i; 0 <= i < strlen(from) ==> to[i] == from[i];
7       ensures \result == to;
8   */
9   char *
10  strcpy(char *to, const char *from)
11  {
12    char *save = to;
13
14    //@ ghost char *origFrom = from;
15    /*@ loop assigns save[0..], to, from;
16        loop variant strlen(from);
17        loop invariant \base_addr(to) == \base_addr(save);
18        loop invariant \base_addr(from) == \base_addr(origFrom);
19        loop invariant 0 <= (from-origFrom) <= strlen(origFrom);
20        loop invariant (to-save) == (from - origFrom);
21        loop invariant \valid(to) && \valid(from);
22        loop invariant \forall integer k; 0 <= k < (from-origFrom) ==> origFrom[k] != 0;
23        loop invariant \forall integer k; 0 <= k < (from-origFrom) ==> save[k] == origFrom[k];
24        loop invariant \forall integer k; (from-origFrom) <= k < strlen(origFrom) ==>
25          origFrom[k] == \at(from[k], Pre);
26    */
27    for (; (*to = *from) != '\0'; ++from, ++to);
28      return(save);
29  }
```

Listing 5. *Strcpy function and its contracts.*

4.3.5.3 Proof Results

All proof obligations are proven by Z3 and Simplify. Z3 proves all except one proof obligation:

```
Default behavior: 30/31, safety: 8/8
```

Simplify proves the missing proof obligation which is for preservation of a loop invariant.

4.3.5.4 Specification Details

The function requires that the buffer *to* is valid and is big enough to hold the contents of the string *from*. The function also requires that the string *source* is a null-terminated, valid string. The function writes to the destination buffer.

The pre-conditions at line 2 and 3 express that the function requires that the buffer and source string do not overlap. For example, without this requirement, the second string could get overwritten. This is an important aspect of proving the post-conditions.

The post-condition at line 6 states that all the characters in the source string including the null character are copied. The last post-condition expresses that the function always returns the pointer to the start of the destination string.

The loop modifies the formal parameters and the loop invariants state that the pointer *from* is never incremented past the end of the original string *from*; at any stage during the loop iteration, the pointers *to* and *from* have been incremented the same amount of times and the pointers *to* and *from* remain valid as they are incremented.

The loop invariant at line 23 expresses that at any stage during the loop iteration, the elements visited so far are identical as a result of the copy operation. This invariant is not enough by itself to prove the first post-condition so the loop invariant at line 24 was added to re-state that the contents of the string *from* are not altered during the loop.

4.3.6 Strcat

The *strcat* and *strncat* functions append a copy of the null-terminated string *append* to the end of the null-terminated string *s*, and then add a terminating '\0'. The string *s* must have sufficient space to hold the result. The *strcat* returns the pointer *s*.

4.3.6.1 Contracts

First version of contracts is as follows:

```
/*@ requires valid_string(s1) && valid_string(s2);
    assigns \nothing;
    ensures \result == \null;
    behavior normal:
       assumes \valid_range(s1, 0, strlen(s1) + strlen(s2));
       assigns s1;
       ensures \forall integer i; 0 <= i < \at(strlen(s1), Old) && s1[i] == \old(s1[i]) && \result == s1 &&
         \forall integer j; \old(strlen(s1)) <= j < strlen(s1) && s1[j] == s2[j];
*/
char *strcat(char *s1, const char *s2);
```

The version based on the man file is as follows:

```
/*@ requires valid_string(s1) && valid_string(s2) && \valid_range(s1, 0, strlen(s1) + strlen(s2));
    assigns s1;
    ensures strlen(s1) == \old(strlen(s1) + strlen(s2));
    ensures \forall integer i; 0 <= i < \at(strlen(s1), Old) && s1[i] == \old(s1[i]);
    ensures \forall integer j; \old(strlen(s1)) <= j < strlen(s1) && s1[j] == s2[j];
    ensures \result == s1;
*/
char *strcat(char *s1, const char *s2);
```

In the first version, the specification written based on author's experience with other implementations of this function, the function returned *null* in the error case. Although this does not match the manual as the function always returns the first string and is conformant with the standard.

The code and the final contracts are given in Listing 6.

```
/*@
  requires valid_string(s);
  requires valid_string(append);
  requires disjoint_strings(s, append);
  requires disjoint_strings_len(s, append, strlen(s) + strlen(append));
  requires \valid_range(s, 0, strlen(s) + strlen(append));
  assigns s[..];
  ensures strlen(s) == strlen{Old}(s) + strlen(append);
  ensures \forall integer i; 0 <= i < strlen{Old}(s) ==> s[i] == \old(s[i]);
  ensures \forall integer j; strlen{Old}(s) <= j < strlen(s) ==>
    s[j] == append[j-strlen{Old}(s)];
  ensures  \result == s;
*/
char *
strcat(char *s, const char *append)
{
  char *save = s;
  /*@ loop assigns s;
      loop invariant 0 <= (s-save) <= strlen(save);
      loop invariant \valid(s);
      loop invariant \base_addr(s) == \base_addr(save);
      loop invariant \forall integer k; 0 <= k < (s-save) ==> save[k] != 0;
  */

  for (; *s; ++s);

  //@ assert *s == '\0' && s == save + strlen(save);
  //@ assert \valid_range(s, 0, strlen(append));
  //@ ghost char *s_cat = s;
  //@ ghost char *origAppend = append;

  //@ assert \valid_range(s_cat, 0, strlen(append)); // prove pli 4 and 8 below.

  /*@
    loop assigns s, save[s_cat-save..], append;
    loop invariant \base_addr(s) == \base_addr(s_cat);
    loop invariant \base_addr(append) == \base_addr(origAppend);
    loop invariant \valid_range(s_cat, 0, strlen{Pre}(append));
    loop invariant (append-origAppend) == (s - s_cat);
    loop invariant 0 <= (append-origAppend) <= (strlen(origAppend)) + 1;
    loop invariant \forall integer k; 0 <= k < (append-origAppend)  ==> origAppend[k] != 0;
    loop invariant \forall integer k; 0 <= k < (append-origAppend) <= strlen(origAppend) ==>
      s_cat[k] == origAppend[k];
    loop invariant \forall integer k; 0 <= k < (append-origAppend) ==>
      save[k + s_cat-save] != 0;
    loop invariant \forall integer k; (append-origAppend) <= k <= strlen(origAppend) ==>
      origAppend[k] == \at(append[k], Pre);
    loop invariant \forall integer k; 0 <= k < (s_cat-save) ==> save[k] == \at(s[k], Pre);
  */
  while ((*s++ = *append++) != '\0');

  //@ assert s_cat[append-origAppend - 1] == 0;
  //@ assert strlen(s_cat) == append-origAppend -1;
  //@ assert strlen(origAppend) == append-origAppend -1;
  //@ assert strlen(s_cat) == strlen(origAppend);

  return(save);
}
```

Listing 6. *Strcat function and its contracts.*

4.3.6.2 Man Version versus Final Version

The parameter names do not match the man file so they had to be renamed in the final version.

The pre-condition for disjoint strings was added as otherwise the function behavior is not safe if the strings overlap.

Also, the following correction was made as the function changes the contents of the buffer not the buffer itself.

```
<<< assigns s1;
>>> assigns s[..];
```

There is a danger of buffer overrun with this function unless the buffer is big enough to hold the result. This is covered by the pre-condition for the function at line 6. The function also guarantees that if all pre-conditions are met then the destination string will also be null-terminated

The man page states that the function adds a terminating null character explicitly. This is probably due to the fact that the man page is written for both *strcat* and *strncat* functions; it is not the best approach to mix documentation for two different functions.

In the final version, the post-condition was re-written more accurately using the state label for logic function at line 8.

In addition, the following mistake was corrected as the index values were not correct for the second string *s2*, as a result the post-condition could not be proven:

```
<<< ensures \forall integer j; \old(strlen(s1)) <= j < strlen(s1) && s1[j] == s2[j];
>>> ensures \forall integer j; strlen{Old}(s) <= j < strlen(s) ==> s[j] == append[j-strlen{Old}(s)];
```

4.3.6.3 Proof Results

All proof obligations were proven by Z3 and Simplify. Z3 proved all except one proof obligation:

```
Default behavior: 77/78, Safety: 10/10
```

Simplify proved the missing proof obligation which was for preservation of a loop invariant.

4.3.6.4 Specification Details

The function requires valid strings, strings that are not null and are null-terminated, as input. The function also assumes that the string *s* contains sufficient valid buffer after the null character to hold the string *append*.

The pre-conditions at lines 4 and 5 state that the function requires that the strings are disjoint. For example, passing the same string is not allowed. Also, the part of the buffer where the second string is copied to should be disjoint too. For example, if the second string is located after the first string in memory, it can get overwritten. This is an important aspect to proving the post-conditions.

In the post condition at line 8, the function ensures that the new length of the resultant string is the old length of the string *s* plus the length of the string *append*. The next post-condition at line 9 expresses the fact that characters of the old string are not modified. The rest of the buffer following the length of the original string, including the null character is overwritten by the characters from the string *append* and this is expressed by the post-condition at line 10.

The assertion after the first loop at line 27 states that at the end of the loop, the pointer *s* should be pointing at the null-terminator and the following expression should be true:

```
(s == save + strlen(save))
```

The assertion was added to help understand why some proof obligations were not initially proven. The assertion on the next line is to verify that at the end of the first loop, the buffer is big enough to hold the contents of the second string. This assertion is essential to prove the loop invariants at lines 38 and 42.

Loop invariants at line 39 and 42 express that at any point during the iteration, pointer *append* has been incremented the same times as the pointer *s* and the characters from the original string *append* have been concatenated to the string *s_cat* that points at the end of the string *s*. However, these two loop invariants are not enough to prove the post-conditions so the following loop invariants at lines 46 and 48 were added to express that the original string *s* and the string *append* are not modified during the loop. This aspect is important to prove that the post-conditions hold.

The assertions at the end of the functions verify that the string is null-terminated and they also help prove the post-condition at line 8.

4.3.7 Strrchr

The *strrchr* function locates the last occurrence of the character *c* in the string *s*. The terminating null character is considered to be part of the string. If *c* is '\0', *strrchr* locates the terminating null character. The *strrchr* function returns a pointer to the located character or *null* if the character does not appear in the string.

4.3.7.1 Contracts

First version of contracts was as follows:

```
/*@
  @ requires valid_string(s);
  @ assigns \nothing;
  @ ensures \exists integer i; 0 <= i <= strlen(s) && s[i] == n &&
  @  (\forall integer j; i < j <= strlen(s) ==> s[j] != n) ==> \result == s+i;
  @ ensures \forall integer i; 0 <= i <= strlen(s) && s[i] != n ==> \result == \null;
*/
char *strrchr(const char *s, int n);
```

The version based on man file was as follows:

```
/*@
  @ requires valid_string(s);
  @ assigns \nothing;
  @ ensures \exists integer i; 0 <= i <= strlen(s) && s[i] == c &&
  @   (\forall integer j; i < j <= strlen(s) ==> s[j] != c) ==> \result == s+i;
  @ ensures \forall integer i; 0 <= i <= strlen(s) && s[i] != c ==> \result == \null;
*/
char *strrchr(const char *s, int c);
```

There is no difference to note between the first and the second versions.

The code and the final contracts are given in Listing 7.

4.3.7.2 Man Version versus Final Version

The final version uses behaviours because it makes proving the generated proof obligations and narrowing down any possible issues easier. The second version matches the manual. However, similarly to *strchr* function, the same documentation bug is found in this function. Therefore, behaviours *b1* and *b2* were added to the final version. Although the manual states the following:

```
"The terminating NUL character is considered part of the string.  If c is `\0', strchr() locates the
terminating '\0'."
```

The code states otherwise and when *c* is '\0', the function returns *null*.

```
/*@ requires valid_string(s);
    assigns \nothing;
    behavior b1:
      assumes c == '\0';
      ensures \result == \null;
    behavior b2:
      assumes strlen(s) == 0;
      ensures \result == \null;
    behavior b3:
      assumes c != '\0' && strlen(s) > 0;
      ensures \exists integer i; 0 <= i < strlen(s) && s[i] == c &&
        (\forall integer j; i < j < strlen(s) ==> s[j] != c) ==> \result == s+i;
    behavior b4:
      assumes c != '\0' && strlen(s) > 0;
      assumes \forall integer i; 0 <= i < strlen(s) && s[i] != c;
      ensures \result == \null;
*/
char *
strrchr(const char *s, int c)
{
  char *t = NULL;
  //@ assert \forall integer i; 0 <= i < strlen(s) ==> s[i] != 0;
  //@ ghost char *orig = s;
  /*@ loop assigns s, t;
      loop invariant \valid(s);
      loop invariant \base_addr(s) == \base_addr(orig);
      loop invariant 0 <= (s-orig) <= strlen(orig);
      loop invariant \forall integer k; 0 <= k < (s-orig) ==> orig[k] != 0;
      loop invariant t == \null || *t == c;
      for b1: loop invariant t == \null;
      for b2: loop invariant t == \null;
      for b3: loop invariant (\exists integer k; 0 <= k < (s-orig) && orig[k] == c)  ==> t != \null;
      for b4: loop invariant t == \null;
  */
  while (*s) {
    if (*s == c)
      t = (char *)s;
    s++;
  }
  return (t);
}
```

Listing 7. *Strrchr and its contracts.*

In addition, the following change was made for the same reason as the comparison is done up to the null character:

```
<<< ensures \exists integer i; 0 <= i <= strlen(s) && s[i] == c
>>> ensures \exists integer i; 0 <= i < strlen(s) && s[i] == c
```

Similarly, the following post-condition was corrected:

```
<<< ensures \forall integer i; 0 <= i <= strlen(s) && s[i] != c ==> \result == \null;
>>> ensures (\forall integer i; 0 <= i < strlen(s) ==> s[i] != c) ==> \result == \null;
```

4.3.7.3 Proof Result

All proof obligations are proven by Simplify, Alt-Ergo and Z3. Simplify proves all proof obligations except two safety proof obligations:

```
b1: 2/2, b2: 2/2, b3: 4/4, b4: 2/2, default behavior: 21/21, safety: 2/4
```

The missing two safety proof obligations are proven by both Alt-Ergo and Z3.

4.3.7.4 Specification Details

The pre-condition of the function states that the string should be a valid, null-terminated string and the frame condition states that the function does not modify anything.

The post-condition at line 11 states that if the character is found in the string and it is the first occurrence from the end of the buffer, then the result will be a pointer to that character.

The post-condition at line 16 states that if the character is not found in the string then the result is *null*.

Within the loop, the temporary pointer *t* is either *null* or points to character *c* in the buffer. Note that the value will be overwritten with the next occurrence found if there is more than one occurrence as the function tries to find the last occurrence. For behaviors *b1*, *b2* and *b4*; *t* will always be *null* as those behaviours are for the case when the character is not part of the string. For behavior *b3*, the loop invariant at line 32 helps prove that the return value is not *null*.

4.3.8 Strstr

The *strstr* function locates the first occurrence of the null-terminated string *little* in the null-terminated string *big*. If *little* is an empty string, then *big* is returned; if *little* occurs nowhere in *big*, *null* is returned; otherwise a pointer to the first character of the first occurrence of *little* is returned.

4.3.8.1 Contracts

First version of the contracts was as follows:

```
/*@
  requires valid_string(s1) && valid_string(s2);
  assigns \nothing;
  ensures strlen(s1) < strlen(s2) ==> \result == \null;
  ensures strlen(s1) >= strlen(s2) && \exists integer i; 0 <= i <= (strlen(s1) - strlen(s2)) &&
    \forall integer k; i <= k <= strlen(s2) && s1[k] == s2[k] ==> \result == s1 + i;
  ensures strlen(s1) >=  strlen(s2) && \forall integer i; 0 <= i <= (strlen(s1) - strlen(s2)) &&
    \exists integer k; i <= k <= strlen(s2) && s1[k] != s2[k] ==> \result == \null;
*/
char *strstr(const char *s1, const char *s2);
```

The second version based on man file was as follows:

```
/*@
  requires valid_string(big) && valid_string(little);
  assigns \nothing;
  ensures strlen(little) == 0 ==> \result == big;
  ensures strlen(big) < strlen(little) ==> \result == \null;
  ensures strlen(big) >= strlen(little) && \exists integer i; 0 <= i <= (strlen(big) - strlen(little)) &&
    contains_string_at_i(big, little, i) && \forall integer k; 0 <= k < i &&
    !contains_string_at_i(big, little, k) ==> \result == big + i;
  ensures strlen(big) >= strlen(little) && \forall integer i; 0 <= i <= (strlen(big) - strlen(little)) &&
    !contains_string_at_i(big, little, i) ==> \result == \null;
*/
char *strstr(const char *big, const char *little);
```

In the first version, the specifications were too weak to prove and did not cover all of the behaviour of the function. Several post-conditions were missing and therefore were added to the second version as described in the man file. For example, the following post-condition was added to cover the empty string case:

```
ensures strlen(little) == 0 ==> \result == big;
```

In order to simplify the specifications and make them more readable, the following predicate was written to check whether a string contains another string starting at index *i*:

```
/*@ predicate contains_string_at_i{L}(char *big, char *little, integer i) =
      \forall integer k; 0 <= k && k < strlen(little) && (k + i) < strlen(big) && big[k + i] == little[k];
*/
```

The code and the final version of the contracts are given in Listing 8.

4.3.8.2 Man Version versus Final Version

The parameter names did not match the manual and therefore were renamed in the final version.

The final specifications use behaviours which are more precise for relatively more complex behaviours of functions like *strstr*. This change helps tracking problems with proving the verification conditions. Several conditions overlap so *assumes* clauses of the behaviours make it easier to prove the relevant post-conditions. For example, the *assumes* clause of behaviour *b5* states that the length of string *find* should be greater than zero and this is important to prove the corresponding post-condition.

4.3.8.3 Proof Results

Z3 proves all proof obligations except two for the default behaviour and one safety proof obligation:

```
b1: 3/3, b2: 3/3, b3: 3/3, b4: 3/3, b5: 3/3, default behaviour: 28/30, safety: 12/13
```

Simplify proves the missing proof obligation for loop invariant preservation. One assertion and a safety proof obligation for *strncmp* function call are not proven.

4.3.8.4 Specification Details

As mentioned earlier, the predicate *contains_string_at_i* was written and used to simplify the specifications and make them more readable. This predicate checks whether a string contains another string starting at index *i* and returns *true* if all the elements of string *big* match the elements of string *little* for the length of string *little* starting at index *i*.

The pre-conditions of the function state that the strings *s* and *find* are required to be valid, null-terminated strings. The frame condition states that the function does not modify the buffers.

Behaviour *b1* states that if the string *find* is empty, then the function will just return the string *s*.

Behaviour *b2* states that if the string *find* is not empty but the string *s* is empty or shorter than the length of string *find* then the function will return *null* as it is not possible that the string *s* contains string *find*.

Similarly, behaviour *b3* states that if the first character of string *find* is not found in string *s* then again it is not possible that the string *s* contains string *find* and the function will return *null*.

Behaviour *b4* states that if the string *s* is longer than the string *find* and the string *s* contains string *find* then the function will return the first occurrence of string *find* in string *s*. The string *find* is only searched up to the index *(strlen(s)-strlen(find))* as higher indexes cannot contain the string *find*. The post-condition ensures that the result of the function will point at the index where the string *find* can be found.

```
1   /*@
2     requires valid_string(s);
3     requires valid_string(find);
4     assigns \nothing;
5     behavior b1:
6       assumes strlen(find) == 0;
7       ensures \result == s;
8     behavior b2:
9       assumes strlen(s) == 0 || strlen(s) < strlen(find);
10      assumes strlen(find) > 0;
11      ensures \result == \null;
12    behavior b3:
13      assumes strlen(s) > 0;
14      assumes strlen(find) > 0;
15      assumes \forall integer i; 0 <= i < strlen(s) && s[i] != *find;
16      ensures \result == \null;
17    behavior b4:
18      assumes strlen(s) >= strlen(find) > 0;
19      assumes \exists integer i; 0 <= i <= (strlen(s) - strlen(find)) && contains_string_at_i(s, find, i)
20        && \forall integer j; 0 <= j < i ==> !contains_string_at_i(s, find, j);
21      ensures contains_string_at_i(\result, find, 0);
22    behavior b5:
23      assumes strlen(s) >= strlen(find) && strlen(find) > 0;
24      ensures \forall integer i; 0 <= i < strlen(s) && !contains_string_at_i(s, find, i) ==>
25        \result == \null;
26  */
27  char *
28  strstr(const char *s, const char *find)
29  {
30    char c, sc;
31    size_t len;
32    //@ ghost char *origFind = find;
33    if ((c = *find++) != 0) {
34      len = strlen(find);
35      //@ ghost char *orig = s;
36      /*@
37        loop invariant \base_addr(s) == \base_addr(orig);
38        loop invariant 0 <= s - orig <= strlen(orig);
39        loop invariant \forall integer k; 0 <= k < (s-orig) ==> orig[k] != 0;
40        loop invariant \forall integer k; 0 <= k < (s-orig) ==> !contains_string_at_i(orig, origFind, k);
41      */
42      do {
43        //@ ghost char *p = s;
44        /*@ loop assigns s;
45            loop invariant \base_addr(s) == \base_addr(orig);
46            loop invariant \base_addr(s) == \base_addr(p);
47            loop invariant \valid(s);
48            loop variant strlen(s);
49            loop invariant 0 <= s - orig <= strlen(orig);
50            loop invariant \forall integer j; 0 <= j < (s-p) ==> orig[j + p - orig] != 0;
51            loop invariant \forall integer j; 0 <= j < (s-p) ==> orig[j + p - orig] != c;
52        */
53        do {
54          if ((sc = *s++) == 0)
55            return (NULL); /*@ assert \forall integer k; 0 <= k < strlen(orig) ==>
56                                 !contains_string_at_i(orig, origFind, k); */
57        } while (sc != c);
58      } while (strncmp(s, find, len) != 0);
59      s--;
60      //@ assert *s == c;
61      //@ assert contains_string_at_i(s, origFind, 0);
62      //@ assert contains_string_at_i(orig, origFind, s-orig);
63    }
64    return ((char *)s);
65  }
```

Listing 8. *Strstr and its contracts.*

Behaviour *b5* states that the function will also return *null* if the string *s* is longer than string *find* and the string *find* is not empty but the string *s* does not contain the string *find*.

The loop invariant at line 40 states that at any stage in the loop iteration, the string *find* is not contained at previous index values visited. This is necessary to prove the post-conditions. Note that the pre-conditions and post-conditions of function *strncmp* as discussed in section 4.3.3 hold during the loop.

At any stage during the inner loop iteration that starts at line 53, the characters at previous index values so far are not equal to the first character of string *find*. Note that this is not stated from the start of the string, as this might not be true. This is stated rather starting from the index value from which the inner iteration starts. The loop variant guarantees termination when the length string *s* is zero.

The assert statement at line 55 is for the case where the function returns *null* from within the loop and verifies that the end of the string *s* is hit and for each index starting from 0 to the length of *s* the string *find* is not contained.

The assertions at the end of the function make sure that when the function returns, the pointer *s* points to the string *find* at index zero and in the original buffer, the string *find* can be found at the index pointer *s* is pointing to. The assertion at line 61 is not proven.

The pre-condition for *strncmp* function call generates two proof obligations. One of them is not proven even though string *s* always remains as a valid string. This might also be the reason why the assertion mentioned above also fails.

4.3.9 Strncpy

The *strncpy* function copies the string *src* to *dst* including the terminating '\0' character. The *strncpy* function does not copy more than *len* characters into *dst* and appends '\0' characters if string *src* is less than *len* characters long. The resulting string *dst* is not terminated if the length of *src* is greater than or equal to *len*. The *strncpy* function only null-terminates the destination string when the length of the source string is less than the length parameter. Finally, the *strncpy* function always returns *dst*.

4.3.9.1 Contracts

The first version of the contracts was as follows:

```
/*@ requires valid_string(s1) && valid_string(s2);
    assigns s1;
    behavior normal:
      assumes strlen(s1) >= minimum(strlen(s2), n);
      ensures \forall integer i; 0 <= i < minimum(strlen(s2), n) && s1[i] == s2[i] && \result == s1;
    behavior error:
      assumes strlen(s1) < minimum(strlen(s2), n);
      assigns \nothing;
      ensures \result == \null;
*/
char *strncpy(char *s1, const char *s2, size_t n)
  __attribute__ ((__bounded__(__string__,1,3)));
```

The second version based on the man page was as follows:

```
/*@ requires valid_string(dst) && valid_string(src);
    requires \valid_range(dst, 0, len);
    assigns dst[0..len];
    ensures \forall integer i; 0 <= i < minimum(len, strlen(src)) && dst[i] == src[i];
    ensures strlen(src) < len ==> \forall integer i; strlen(src) <= i < len && dst[i] == '\0';
    ensures \result == dst;
*/
char *strncpy(char *dst, const char *src, size_t len)
  __attribute__ ((__bounded__(__string__,1,3)));
```

The first version assumed that the function returned *null* and modified nothing for the error condition where the destination string was shorter but this did not match the man file. If one considers the *strchr* function, the string functions as a set do not behave consistently and this makes them harder to learn.

The code and the final contracts are given in Listing 9 and Listing 10.

4.3.9.2 Man Version versus Final Version

The parameter name *n* does not match the man file which uses *len*, therefore it was renamed in the final specifications.

The destination *dst* does not have to be a valid string, it needs to be a valid buffer and this was fixed in the final version.

```
1   /*@ requires valid_string(src);
2       requires \valid_range(dst, 0, n);
3       requires disjoint_strings(dst, src);
4       requires disjoint_strings_len(dst, src, n);
5       ensures \result == dst;
6       behavior b1:
7         assumes n == 0;
8         assigns \nothing;
9       behavior b2:
10        assumes n > 0 && strlen(src) > n;
11        assigns dst[..];
12        ensures \forall integer i; 0 <= i < n - 1 ==> dst[i] == src[i];
13      behavior b3:
14        assumes n > 0 && strlen(src) < n;
15        assigns dst[0..];
16        ensures \forall integer i; 0 <= i <= strlen(src) ==> dst[i] == src[i];
17        ensures \forall integer i; strlen(src) < i < n ==> dst[i] == 0;
18      behavior b4:
19        assumes n > 0 && strlen(src) == n;
20        assigns dst[0..];
21        ensures \forall integer i; 0 <= i < strlen(src) ==> dst[i] == src[i];
22  */
23  char *
24  strncpy(char *dst, const char *src, size_t n)
```

Listing 9. *Strncpy function and its contracts.*

The man file does not mention what happens if the destination buffer is shorter than the count parameter but this is an important requirement for this function and was added as a pre-condition at line 2.

In the final version, the post-conditions correctly use implication *(==>)* instead of conjunction *(&&)* so the case of empty string *src* or *n* being zero can be proven.

The final specification uses behaviours to be more precise for relatively more complex functions like this one. Separating the post-conditions according to different behaviours simplifies the specifications and the proof. The final version also separates the *requires* clauses to their own lines. These changes help tracking issues while proving the verification conditions of this function and its callers.

The behaviour *b1* also covers the case where *n* is zero or the source string is empty where the function does not modify anything. This is not mentioned in the manual.

4.3.9.3 Proof Results

Z3 proves all proof obligations except 3 of them:

```
b1: 6/6, b2: 9/9, b3: 12/12, b4: 9/9, default behaviour: 89/92, safety: 14/14
```

Simplify proves one missing proof obligation for a loop invariant preservation and Alt-Ergo proves one missing proof obligation for an assertion.

```
char *
strncpy(char *dst, const char *src, size_t n)
{
  if (n != 0) {
    char *d = dst;
    const char *s = src;
    /*@ loop assigns d, s, n, dst[..];
        loop invariant \at(n, Pre) >= n > 0;
        loop invariant strlen(src) >= \at(n, Pre) ==> 0 <= d-dst <= \at(n, Pre) <= strlen(src);
        loop invariant strlen(src) >= \at(n, Pre) ==> 0 <= s-src <= \at(n, Pre) <= strlen(src);
        loop invariant strlen(src) >= \at(n, Pre) ==> d-dst == s-src;
        loop invariant strlen(src) < \at(n, Pre) ==> d-dst <= strlen(src) ==> d-dst == s-src;
        loop invariant strlen(src) < \at(n, Pre) ==> 0 <= s-src <= strlen(src) < \at(n, Pre);
        loop invariant strlen(src) < \at(n, Pre) ==> 0 <= d-dst <= strlen(src) < \at(n, Pre);
        loop invariant d-dst == \at(n, Pre) - n;
        loop invariant \base_addr(d) == \base_addr(dst);
        loop invariant \base_addr(s) == \base_addr(src);
        loop invariant \valid(d) && \valid(s);
        loop invariant \valid_range(dst, 0, \at(n, Pre));
        loop invariant \valid_range(src, 0, strlen(src));
        loop invariant strlen(src) >= \at(n, Pre) ==> (\forall integer k; 0 <= k < (d-dst) < \at(n, Pre)
          ==> dst[k] == src[k]);
        loop invariant strlen(src) < \at(n, Pre) ==> \forall integer k; 0 <= k < (d-dst) ==>
          dst[k] == src[k];
        loop invariant \forall integer k; 0 <= k < (s-src) <= strlen(src) ==> src[k] != 0;
        loop invariant \forall integer k; 0 <= k <= strlen(src) ==> src[k] == \at(src[k], Pre);
    */
    do {
      if ((*d++ = *s++) == 0){
        /* NUL pad the remaining n-1 bytes */
        //@ ghost char *p = d;
        //@ ghost size_t n2 = n;
        //@ assert *(p-1) == 0;
        //@ assert strlen(src) == p - dst - 1;
        //@ assert dst[strlen(src)] == 0;
        //@ assert \forall integer i; 0 <= i < (p-dst) ==> dst[i] == src[i];
        /*@ loop assigns d, n, dst[p-dst..];
            loop invariant strlen(src) < \at(n, Pre) ==> \valid(d);
            loop invariant \base_addr(d) == \base_addr(dst);
            loop invariant \base_addr(d) == \base_addr(p);
            loop invariant \at(n, Pre) >= n2 >= n > 0;
            loop invariant 0 <= d - p <= (n2-1);
            loop invariant d - p == n2 - n;
            loop invariant 0 <= d-dst <= \at(n, Pre);
            loop invariant \forall integer k; (p-dst) <= k < (d-dst) ==> dst[k] == 0;
            loop invariant \forall integer k; 0 <= k <= strlen(src) ==> src[k] == \at(src[k], Pre);
            loop invariant \forall integer i; 0 <= i < (p-dst) ==> dst[i] == src[i];
        */
        while (--n != 0)
          *d++ = 0;
        break;
      }
    } while (--n != 0);
  }
  return (dst);
}
```

Listing 10. *Strncpy function body and its contracts.*

4.3.9.4 Specification Details

The pre-conditions of the function state that the source string is required to be a valid string and the destination needs to be a valid buffer of length at least *n*.

The post-condition at line 5 is common to all behaviours and states that the return value is always a pointer to the destination string.

Behaviour *b1* states that if the formal parameter *n* is 0, then the function will not modify the destination buffer. Note that if the source string is empty, the destination buffer is still modified.

Behaviour *b2* states that the function will modify the destination buffer and copy *n* characters from the source string into the destination string if the formal parameter *n* is greater than zero and the length of the source string is greater than parameter *n*. Note that this behaviour does not mention the length of destination string in post-conditions as the string is not null-terminated as explained in the man file.

Behaviour *b3* applies if the formal parameter *n* is greater than zero and the length of the source string is less than the parameter *n*. In this case, the function will modify the destination buffer by copying the source string and appending a null character to the remaining locations up to location *n*.

Behaviour *b4* covers the case where the length of the source string is equal to *n*. In this case, all the characters from the source string will be copied without null-terminating the resultant string as explained in the man file.

Loop invariant at line 30 states that value of *n* is decremented from the value passed into the function down to zero. The part *n>0* is important for proving the safety proof obligations for overflow.

The loop invariants below express that if the length of source string is equal to or greater than the original value of *n*, then the pointer *s* and *d* are incremented together during the loop, up to the value of the count:

```
loop invariant strlen(src) >= \at(n, Pre) ==> 0 <= d-dst <= \at(n, Pre) <= strlen(src);
loop invariant strlen(src) >= \at(n, Pre) ==> 0 <= s-src <= \at(n, Pre) <= strlen(src);
loop invariant strlen(src) >= \at(n, Pre) ==> d-dst == s-src;
```

The loop invariants below express that if the length of source string is less than the original value of *n*, then the pointer *s* and *d* are incremented up to the end of string *src* instead. Note that *d-dst* equals to *s-src* only if *d* is not past the end of the string:

```
loop invariant strlen(src) < \at(n, Pre) ==> d-dst <= strlen(src) ==> d-dst == s-src;
loop invariant strlen(src) < \at(n, Pre) ==> 0 <= s-src <= strlen(src) < \at(n, Pre);
loop invariant strlen(src) < \at(n, Pre) ==> 0 <= d-dst <= strlen(src) < \at(n, Pre);
```

The loop invariant at line 37 is an important loop invariant as it states that the pointer *d* is never incremented more than the value of original *n*.

The two loop invariants below state that the characters of the source string are copied one by one to the destination buffer. The first one also states the fact that the copy is done up to the original value of *n*:

```
loop invariant strlen(src) >= \at(n, Pre) ==>
  (\forall integer k; 0 <= k < (d-dst) < \at(n, Pre) ==> dst[k] == src[k]);
loop invariant strlen(src) < \at(n, Pre) ==> \forall integer k; 0 <= k < (d-dst) ==> dst[k] == src[k];
```

The inner loop appends null characters for the remainder of the buffer. The use of ghost variables around line 53 allows holding the values of pointer *d* and the variable *n* just before the inner loop is entered.

The assertions after the first loop help prove all the post-conditions. The assertion at line 58 verifies that the source string has been fully copied including the null character and therefore the length of the destination string is the same as source string before the inner loop.

The loop assigns at line 59 states that the loop increments pointer *d*, decrements *n* and modifies the buffer starting from *p-dst*. Note that these can be better expressed in ACSL but the version of the tools used at the time did not support this.

One proof obligation generated for the loop invariant at line 67 is the only proof obligation that was not proven for this function.

4.3.10 Strlcpy

The *strlcpy* function copies strings. It is designed to be a safer, more consistent, and less error prone replacement for *strncpy* function. Unlike that function, *strlcpy* takes the full size of the buffer, not just the length and guarantees to null-terminate the result as long as *size* is larger than zero. Note that a byte for the null character should be included in the size. Also note that *strlcpy* only operates on true "C" strings. This means that *src* must be null-terminated. The strlcpy function copies up to *size – 1* characters from the null-terminated string *src* to *dst*, null-terminating the result. The *strlcpy* returns the total length of the string it tried to create therefore returns the length of *src*.

4.3.10.1 Contracts

The version based on man file was as follows:

```
/*@ requires \valid_range(dst, 0, size) && valid_string(src);
    behavior b1:
      assumes size == 0 || strlen(src) == 0;
      assigns \nothing;
      ensures \result == 0;
    behavior b2:
      assumes size > 0 && strlen(src) > 0;
      assigns dst[0..size];
      ensures \forall integer i; 0 <= i < minimum(size, strlen(src)) ==> dst[i] == src[i];
      ensures dst[size] == 0;
      ensures \result == strlen(src);
*/
size_t strlcpy(char *dst, const char *src, size_t size)
  __attribute__ ((__bounded__(__string__,1,3)));
```

The code and the final contracts are shown in Listing 11 and 12.

4.3.10.2 Man Version versus Final Version

The parameter *size* is named as *siz* in the code; therefore the specifications had to be updated.

The following behaviour was not correct in the man version; if *src* is empty, the buffer is still modified as a null character is copied:

```
behavior b1:
  assumes size == 0 || strlen(src) == 0;
  assigns \nothing;
  ensures \result == 0;
```

Also, regardless, the return value is always length of the *src* string.

Behaviour *b2* was further divided into two behaviours in the final version as using *minimum* logic function complicated the specifications and the proof; depending on which string is smaller, the bounds comparison changed from < to <=.

```
1   /*@ requires \valid_range(dst, 0, siz);
2       requires valid_string(src);
3       requires disjoint_strings(dst, src);
4       requires disjoint_strings_len(dst, src, siz);
5       requires disjoint_strings_len(dst, src, strlen(src));
6       requires strlen(src) < INT_MAX;
7       ensures \result == strlen(src);
8       behavior b0:
9         assumes siz == 0;
10        assigns \nothing;
11      behavior b1:
12        assumes siz >= 1;
13        assumes siz <= strlen(src) + 1;
14        assigns dst[..];
15        ensures \forall integer i; 0 <= i < (siz - 1) ==> dst[i] == src[i];
16        ensures dst[siz - 1] == 0;
17      behavior b2:
18        assumes siz > (strlen(src) + 1);
19        assigns dst[..];
20        ensures \forall integer i; 0 <= i <= strlen(src) ==> dst[i] == src[i];
21  */
22  size_t
23  strlcpy(char *dst, const char *src, size_t siz)
```

Listing 11. *Strlcpy and its contracts.*

In addition, the following clause was corrected as the buffer is not always modified up to *size*:

```
assigns dst[0..size];
```

Finally, note that the man page contains two functions and their descriptions together which is not a good experience.

4.3.10.3 Proof Results

All proof obligations were proven by Z3 except one proof obligation for an assertion:

```
b0: 9/9, b1: 27/27 b2: 18/18, default behaviour: 168/169, safety: 71/71
```

4.3.10.4 Specification Details

The function requires that *src* is a valid, null-terminated string and the destination buffer is a valid buffer of length *siz*. The function also requires that the destination buffer and the source string are disjoint for correctness of the function.

The pre-condition at line 6 was added in order to prove safety proof obligations for overflow.

The post-condition at line 7 is common to all behaviours and states that the function always returns the length of the source string.

Behaviour *b0* states that if the size is zero then the function does not modify the destination buffer.

Behaviour *b1* states that if the size is greater than or equal to one and it is equal to or less than the length of the source string plus one then the function modifies the destination buffer by copying *siz-1* characters from the source string and null-terminating the destination string.

Behaviour *b2* states that if the size is greater than the length of the source string plus one, then the function modifies the destination buffer by copying all of the source string.

Loop invariant at line 32 expresses that variable *n* is decremented from value *siz* to zero. It is important for overflow safety proof obligations that it is expressed as greater than zero.

Loop invariants at line 33 and 34 state that the source string is traversed but never beyond the length of the string. If size is smaller than the length then the string is traversed up to that value.

Loop invariants at line 39 and 40 are important to prove the post-conditions.

Loop invariants at line 41 and 43 state that the source string is copied character by character to corresponding index values in the destination string. First one also expresses the fact that the string is not copied to beyond the index value of *siz*.

```
22 size_t
23 strlcpy(char *dst, const char *src, size_t siz)
24 {
25   char *d = dst;
26   const char *s = src;
27   size_t n = siz;
28   /* Copy as many bytes as will fit */
29   if (n != 0) {
30     /*@
31       loop assigns n, d, s, dst[..];
32       loop invariant 0 < n <= siz;
33       loop invariant siz > (strlen(src) + 1) ==> 0 <= (s-src) <= strlen(src);
34       loop invariant 1 <= siz <= strlen(src) + 1 ==> 0 <= (s-src) < siz <= strlen(src) + 1;
35       loop invariant \base_addr(s) == \base_addr(src);
36       loop invariant \base_addr(d) == \base_addr(dst);
37       loop invariant \valid(d);
38       loop invariant (s-src) <= strlen(src) ==> \valid(s);
39       loop invariant (d-dst) == (s-src);
40       loop invariant (siz - n) == (s-src);
41       loop invariant 1 <= siz <= strlen(src) + 1 ==> \forall integer k; 0 <= k < (s-src) <= siz ==>
42         dst[k] == src[k];
43       loop invariant siz > (strlen(src) + 1) ==> \forall integer k; 0 <= k < (s-src) ==>
44         dst[k] == src[k];
45       loop invariant \forall integer k; 0 <= k < (s-src) <= strlen(src) ==> src[k] != '\0';
46       loop invariant \forall integer k; 0 <= k <= strlen(src) ==> src[k] == \at(src[k], Pre);
47     */
48     while (--n != 0) {
49       if ((*d++ = *s++) == '\0')
50         break;
51     }
52     //@ assert siz > strlen(src) + 1 ==> s-src == strlen(src) + 1;
53     //@ assert siz > strlen(src) + 1 ==> n > 0;
54   }
55
56   /* Not enough room in dst, add NUL and traverse rest of src */
57   if (n == 0) {
58     if (siz != 0)
59       *d = '\0';  /* NUL-terminate dst */
60     //@ assert (d-dst) == (siz - 1);
61     //@ assert siz <= strlen(src) + 1;
62     //@ ghost char *p = s;
63     /*@
64       loop assigns s;
65       loop invariant \base_addr(s) == \base_addr(src);
66       loop invariant \base_addr(s) == \base_addr(p);
67       loop invariant (s-src) <= strlen(src) ==> \valid(s);
68       loop invariant p - src <= s - src <= strlen(src);
69       loop invariant \forall integer k; p - src <= k < (s-src) <= strlen(src) ==> src[k] != '\0';
70       loop invariant \forall integer k; 0 <= k <= strlen(src) ==> src[k] == \at(src[k], Pre);
71     */
72     while (*s++)
73       ;
74   }
75   //@ assert *(s-1) == 0;
76   //@ assert (s - src - 1) == strlen(src);
77   return(s - src - 1);        /* count does not include NUL */
78 }
```

Listing 12. *Strlcpy function body and its contracts.*

A proof obligation generated for the assertion at line 60 was the only proof obligation not proven for this function.

The assertion at line 61 verifies that if the code reaches this point then the value of parameter *siz* passed in must have been less than the length of the source string plus one.

The assertions at the end of the function help prove that the function always returns the length of the source string.

4.3.11 Strncat

The *strncat* function appends a copy of the null-terminated string *append* to the end of the null-terminated string *s* and then adds a terminating '\0'. The string *s* must have sufficient space to hold the result. The *strncat* function appends not more than *count* characters where space for the terminating '\0' should not be included in *count*. *Strncat* will append as many characters from the source string as will fit; or will not append any if there is no space. *Strncat* always returns the pointer *s*.

4.3.11.1 Contracts

The first version of the contracts was as follows:

```
/*@
  requires \valid_range(s1, 0, minimum(n, strlen(s2)) -1) && valid_string(s2);
  assigns s1[0..minimum(n, strlen(s2)) - 1];
  ensures strlen(s1) == \old(strlen(s1)) + minimum(n, strlen(s2));
  ensures \forall integer k; 0 <= k < \old(strlen(s1)) ==> s1[k] == \old(s1[k]);
  ensures \forall integer k; \old(strlen(s1)) <= k < minimum(n, strlen(s2)) ==>
    s1[k] == s2[k-\old(strlen(s1))];
  ensures \result == s1;
*/
char *strncat(char *s1, const char *s2, size_t n)
  __attribute__ ((__bounded__(__string__,1,3)));
```

The second version based on the man file was as follows:

```
/*@
  requires \valid_range(s, 0, minimum(count, strlen(append))) && valid_string(append);
  assigns s[0..minimum(count, strlen(append)) - 1];
  ensures strlen(s) == \old(strlen(s)) + minimum(count, strlen(append)) + 1;
  ensures \forall integer k; 0 <= k < \old(strlen(s)) ==> s[k] == \old(s[k]);
  ensures \forall integer k; \old(strlen(s)) <= k < minimum(count, strlen(append)) ==>
    s[k] == append[k-\old(strlen(s))];
  ensures s[minimum(count, strlen(append)) + 1] == '\0';
  ensures \result == s;
*/
char *strncat(char *s, const char *append, size_t count)
  __attribute__ ((__bounded__(__string__,1,3)));
```

The first version did not cover that the function adds a null character. Man file states this correctly.

The code and the final contracts are shown in Listing 13 and 14.

4.3.11.2 Man Version versus Final Version

The parameter names do not match the man file therefore they were renamed in the final version of the specifications. Note that this can get confusing for comparing the specifications as one uses *s* for the destination string while the other one uses *dst* for destination string and *src* for the source string.

The final specification uses behaviours to be more precise for relatively more complex functions like this one. It is better to separate the post-conditions according to different behaviours. The final version also separates the *requires* clauses to their own lines. These changes help to track issues while proving the verification conditions of this function and its callers.

The second version is more precise about the locations modified in the destination buffer; however using the function *minimum* in the *assigns* clause was not supported by the tools so it was omitted in the final version. In the future versions, this should be possible.

The second version was not requiring the destination to be null-terminated and this was fixed in the final version.

The second version was also wrong about the following where it missed the length of the original string in the valid range required:

```
<<< requires \valid_range(s, 0, minimum(count, strlen(append)))
>>> requires \valid_range(dst, 0, strlen(dst) + minimum(n, strlen(src)) + 1);
```

Finally, the final version uses the state labels for old values correctly:

```
strlen{Old}(dst)
```

4.3.11.3 Proof Results

The following are the results of running Z3:

```
b1: 5/5, b2: 12/13, b3: 11/13, default behaviour: 97/100, safety: 13/14.
```

One of the missing proof obligations for default behaviour is proven by Simplify.

```
1   /*@
2     requires valid_string(src);
3     requires valid_string(dst);
4     requires \valid_range(dst, 0, strlen(dst) + minimum(n, strlen(src)) + 1);
5     requires disjoint_strings(src, dst);
6     requires disjoint_strings_len(dst, src, strlen(dst) + strlen(src) + 1);
7     requires disjoint_strings_len(dst, src, strlen(dst) + n + 1);
8     ensures \result == dst;
9     behavior b1:
10      assumes n == 0;
11      assigns \nothing;
12    behavior b2:
13      assumes n > 0 && strlen(src) <= n;
14      assigns dst[..];
15      ensures strlen(dst) == strlen{Old}(dst) + strlen(src);
16      ensures \forall integer k; 0 <= k < strlen{Old}(dst) ==> dst[k] == \old(dst[k]);
17      ensures \forall integer k; 0 <= k < strlen(src) ==> dst[k + strlen{Old}(dst)] == src[k];
18    behavior b3:
19      assumes n > 0  && strlen(src) > n;
20      assigns dst[..];
21      ensures strlen(dst) == strlen{Old}(dst) + n;
22      ensures \forall integer k; 0 <= k < strlen{Old}(dst) ==> dst[k] == \old(dst[k]);
23      ensures \forall integer k; 0 <= k < n ==> dst[k + strlen{Old}(dst)] == src[k];
24  */
25  char *
26  strncat(char *dst, const char *src, size_t n)
```

Listing 13. *Strncat and its contracts.*

4.3.11.4 Specification Details

The function requires that both strings need to be valid, null-terminated strings. Another pre-condition at line 4 expresses that the destination buffer should be large enough to hold the original string and the minimum of *n* and the length of source string; the null character appended at the end is covered by adding one. The function also requires that the buffers do not overlap as this would lead to undefined behavior. The function modifies the destination buffer and returns the pointer to the destination string.

Behavior *b1* states that if the count passed in is zero then the function will not modify anything. This is something that could be added to man to be more specific about the function's behaviour.

```
char *
strncat(char *dst, const char *src, size_t n)
{
  if (n != 0) {
    char *d = dst;
    const char *s = src;
    /*@ loop assigns d;
        loop invariant \base_addr(d) == \base_addr(dst);
        loop invariant \valid(d);
        loop invariant \valid_range(dst, 0, strlen{Pre}(dst));
        loop invariant 0 <= (d - dst) <= strlen{Pre}(dst);
        loop invariant \forall integer i; 0 <= i < (d-dst) ==> dst[i] != 0;
        loop invariant \forall integer i; 0 <= i <= strlen{Pre}(dst) ==> dst[i] == \at(dst[i], Pre);
        loop invariant \forall integer i; 0 <= i <= strlen(src) ==> src[i] == \at(src[i], Pre);
    */
    while (*d != 0)
      d++;
    //@ assert *d == 0;
    //@ assert d-dst == strlen{Pre}(dst);
    //@ assert d == dst + strlen{Pre}(dst);
    //@ assert \valid_range(d, 0, minimum(n, strlen(src)) + 1);

    /*@ loop assigns d, s, n, dst[..];
        loop invariant \valid(d);
        loop invariant (s-src) <= strlen(src) ==> \valid(s);
        loop invariant \base_addr(d) == \base_addr(dst);
        loop invariant \base_addr(s) == \base_addr(src);
        loop invariant \valid_range(dst, 0, strlen{Pre}(dst) + minimum(\at(n, Pre), strlen(src)) + 1);
        loop invariant \at(n, Pre) >= n > 0;
        loop invariant \at(n, Pre) >= strlen(src) ==> 0 <= (s-src) <= strlen(src);
        loop invariant \at(n, Pre) < strlen(src) ==> 0 <= (s-src) <= \at(n, Pre) < strlen(src);
        loop invariant \at(n, Pre) >= strlen(src) ==> strlen{Pre}(dst) <= (d-dst) <= strlen{Pre}(dst) +
          strlen(src) <= strlen{Pre}(dst) + \at(n, Pre);
        loop invariant \at(n, Pre) < strlen(src)  ==> strlen{Pre}(dst) <= (d-dst) <= strlen{Pre}(dst) +
          \at(n, Pre) < strlen{Pre}(dst) + strlen(src);
        loop invariant (s-src) <= strlen(src) ==> d - dst - strlen{Pre}(dst) ==  (s - src);
        loop invariant d - dst - strlen{Pre}(dst) == (\at(n, Pre) - n);
        loop invariant \forall integer i; strlen{Pre}(dst) <= i < (d-dst) ==>
          dst[i] == src[i - strlen{Pre}(dst)];
        loop invariant \forall integer i; 0 <= i < (s-src) <= strlen(src) ==> src[i] != 0;
        loop invariant \forall integer i; 0 <= i <= strlen(src) ==> src[i] == \at(src[i], Pre);
        loop invariant \forall integer i; 0 <= i < strlen{Pre}(dst) ==> dst[i] == \at(dst[i], Pre);
        loop invariant \forall integer i; strlen{Pre}(dst) <= i < (d-dst) ==> dst[i] != 0 ;
    */
    do {
      if ((*d = *s++) == 0)
        break;
      d++;
    } while (--n != 0);
    *d = 0;
    //@ assert dst[d-dst] == 0;
    //@ assert \at(n, Pre) > strlen(src) ==> *(s-1) == 0;
    //@ assert \at(n, Pre) > strlen(src) ==> strlen(src) == s-src -1;
    //@ assert \at(n, Pre) > strlen(src) ==> strlen(dst) == strlen{Pre}(dst) + s-src -1;
    //@ assert \at(n, Pre) > strlen(src) ==> strlen(dst) == strlen{Pre}(dst) + strlen(src);
    //@ assert \at(n, Pre) < strlen(src) ==> dst[strlen{Pre}(dst) + \at(n, Pre)] == 0;
    //@ assert \at(n, Pre) < strlen(src) ==> d-dst == (strlen{Pre}(dst) + \at(n, Pre)) && *d == 0;
    //@ assert \at(n, Pre) < strlen(src) ==> strlen(dst) == strlen{Pre}(dst) + \at(n, Pre);
    //@ assert \at(n, Pre) == strlen(src) ==> dst[strlen{Pre}(dst) + \at(n, Pre)] == 0;
    //@ assert \at(n, Pre) == strlen(src) ==> d-dst == (strlen{Pre}(dst) + \at(n, Pre)) && *d == 0;
    //@ assert \at(n, Pre) == strlen(src) ==> strlen(dst) == strlen{Pre}(dst) + \at(n, Pre);
  }
  return (dst);
}
```

Listing 14. *Strncat function body and its contracts.*

Behavior *b2* states that if the count is greater than zero and the length of the source string is smaller or less than the count, then the function will modify the buffer and the corresponding post-conditions will hold. The length of the destination string will be its old length plus the length of the source string. Adding the null-terminator is covered here as this post-condition is about the length of the final string which does not include the null character. The original string is preserved and the source string is only appended to the original string.

Behavior *b3* states that if the count is greater than zero and the length of the source string is greater than the count, then the function will modify the buffer and the corresponding post-conditions will hold. The length of the destination string will be its old length plus the count. Adding the null-terminator is covered here as this post-condition is about the length of the final string which does not include null character. The original string is preserved and the source string is only appended to the original string by copying *n* characters.

The first loop just finds the end of the destination string and does not modify anything apart from moving the pointer *d* to the end of the string.

The loop invariant at line 53 states that the value of *n* is decremented from the value passed into the function down to zero. It is important to express that *n* stays greater than zero for proving overflow safety proof obligations.

In the loop invariants below, the pointer *s* is incremented but never beyond the length of the string. If the count is less than the length of the string, then it is incremented only up to the original value of *n*:

```
loop invariant \at(n, Pre) >= strlen(src) ==> 0 <= (s-src) <= strlen(src);
loop invariant \at(n, Pre) < strlen(src) ==> 0 <= (s-src) <= \at(n, Pre) < strlen(src);
```

In the loop invariants below, the pointer *d* starts with pointing at the null character in the original string and is incremented as many times as the length of the source string or the count depending on whichever is smaller:

```
loop invariant \at(n, Pre) >= strlen(src) ==>
  strlen{Pre}(dst) <= (d-dst) <= strlen{Pre}(dst) + strlen(src) <= strlen{Pre}(dst) + \at(n, Pre);
loop invariant \at(n, Pre) < strlen(src)  ==>
  strlen{Pre}(dst) <= (d-dst) <= strlen{Pre}(dst) + \at(n, Pre) < strlen{Pre}(dst) + strlen(src);
```

The following are essential loop invariants to prove proof obligations for the conditional post-conditions at lines 17 and 23:

```
loop invariant (s-src) <= strlen(src) ==> d - dst - strlen{Pre}(dst) ==  (s - src);
loop invariant d - dst - strlen{Pre}(dst) == (\at(n, Pre) - n);
loop invariant \forall integer i; strlen{Pre}(dst) <= i < (d-dst) ==> dst[i] == src[i - strlen{Pre}(dst)];
```

The following loop invariants express that the source string and the original destination string contents are never modified:

```
loop invariant \forall integer i; 0 <= i <= strlen(src) ==> src[i] == \at(src[i], Pre);
loop invariant \forall integer i; 0 <= i < strlen{Pre}(dst) ==> dst[i] == \at(dst[i], Pre);
```

The assertions at the end of the function help prove the conditional post-conditions at lines 15 and 21.

One proof obligation for dereferencing of pointer *d* is not proven.

4.3.12 Strlcat

The *strlcat* function appends the null-terminated string *src* to the end of *dst*. It is designed to be a safer, more consistent, and less error prone replacement for *strncat*. Unlike that function, *strlcat* takes the full size of the buffer, not just the length, and guarantees to null-terminate the result as long as there is at least one byte free in *dst*. Note that a byte for the null character should be included in *size*. Also note that *strlcat* only operates on true "C" strings. This means that both *src* and *dst* must be null-terminated.

Strlcat will append at most *size - strlen(dst) – 1* bytes and will null-terminate the result. *Strlcat* function returns the total length of the string it tried to create. That means the initial length of *dst* plus the length of *src*. While this may seem somewhat confusing, it was done to make truncation detection simple.

Note, however, that if *strlcat* traverses *size* characters without finding a null character, the length of the string is considered to be *size* and the destination string will not be null-terminated since there was no space for the null character. This keeps *strlcat* from running off the end of a string. In practice, this should not happen as it means that either *size* is incorrect or that *dst* is not a proper "C" string. The check exists to prevent potential security problems in incorrect code.

4.3.12.1 Contracts

The version based on man file was as follows:

```
/*@
  requires valid_string(src) && valid_string(dst) && \valid_range(dst, 0, size);
  assigns dst;
  behavior b1:
    assumes size == 0 || strlen(src) == 0;
    assigns \nothing;
    ensures \result == 0;
  behavior b2:
    assumes size > 0 && strlen(src) > 0 && strlen(dst) < size;
    ensures strlen(dst) == \old(strlen(dst)) + minimum(size, strlen(src));
    ensures \forall integer k; 0 <= k < \old(strlen(dst)) ==> dst[k] == \old(dst[k]);
    ensures \forall integer k; 0 <= k < minimum(size, strlen(src)) ==>
      dst[k + \old(strlen(dst))] == src[k];
    ensures dst[strlen(dst)] == '\0';
    ensures \result == \old(strlen(dst)) + strlen(src);
  behavior b3:
    assumes size > 0 && strlen(src) > 0 && strlen(dst) >= size;
    assigns \nothing;
    ensures \result == size + strlen(src);
*/
size_t strlcat(char *dst, const char *src, size_t size)
  __attribute__ ((__bounded__(__string__,1,3)));
```

The code and the final contracts are shown in Listing 15 and 16.

4.3.12.2 Man Version versus Final Version

The way the documentation is mixing two separate functions is not leading to a good experience.

The parameter *size* is named differently in the manual and the code; therefore it was renamed in the final specification.

The final version splits the second behaviour into two behaviours to be able to prove the generated verification conditions. It also merges the final behaviour with the first one.

The final version also moves the common post-conditions in behaviours to a more generic post-condition in default behavior at line 8. Note that this post-condition exactly matches the comment in the source code and therefore is good to prove. However, this does not match exactly what the man page says:

```
"strlcat() function returns the total length of the string it tried to create. For strlcat() that means the
initial length of dst plus the length of src."
```

This statement is only true for the behavior *b2* in the final specifications.

```
/*@
  requires valid_string(src);
  requires valid_string(dst);
  requires \valid_range(dst, 0, siz);
  requires disjoint_strings(dst, src);
  requires disjoint_strings_len(dst, src, siz);
  ensures \forall integer k; 0 <= k < strlen{Old}(dst) ==> dst[k] == \old(dst[k]);
  ensures \result == strlen(src) + minimum(strlen{Old}(dst), siz);
  behavior b0:
    assumes siz == 0 || strlen(dst) >= siz;
    assigns \nothing;
  behavior b1:
    assumes siz > 0 && strlen(dst) < siz;
    assumes 1 == (siz - strlen(dst));
    assigns dst[..];
    ensures strlen(dst) == strlen{Old}(dst);
  behavior b2:
    assumes siz > 0 && strlen(dst) < siz;
    assumes 1 < (siz - strlen(dst));
    assigns dst[..];
    ensures strlen(src) < (siz - strlen{Old}(dst) - 1) ==> (\forall integer k; 0 <= k < strlen(src) ==>
      dst[k + strlen{Old}(dst)] == src[k]);
    ensures strlen(src) >= (siz - strlen{Old}(dst) - 1) ==>
      (\forall integer k; 0 <= k < (siz - strlen{Old}(dst) - 1) ==> dst[k + strlen{Old}(dst)] == src[k]);
    ensures strlen(src) < (siz - strlen{Old}(dst) - 1) ==> strlen(dst) == strlen{Old}(dst) + strlen(src);
    ensures strlen(src) >= (siz - strlen{Old}(dst) - 1) ==> strlen(dst) == siz - 1;
*/
size_t
strlcat(char *dst, const char *src, size_t siz)
```

Listing 15. *Strlcat and its contracts.*

The following post-condition was wrong in the second version as *dst* buffer itself is not modified and was corrected in the final version:

```
<<< ensures \result == strlen(src) + minimum(siz, strlen(\old(dst)));
>>> ensures \result == strlen(src) + minimum(strlen{Old}(dst), siz);
```

The following change in behavior *b1* was also made as the frame condition stating that nothing was modified could not be proven. In fact, the *dst* string is still modified by adding a null-terminator even if it only overwrites the existing null character. As well as removing the empty string case, the case for the destination string length being greater than the size was added:

```
<<< assumes size == 0 || strlen(src) == 0;
>>> assumes siz == 0 || strlen(dst) >= siz;
    assigns \nothing;
```

Consequently, the check for the length of the source string was also removed from the *assumes* clauses of the other behaviours.

Finally, a bigger mistake was made in second version where parameter *siz* was interpreted as the number of characters to be copied but it is rather the full buffer size:

```
  assumes size > 0 && strlen(src) > 0 && strlen(dst) < size;
  ensures strlen(dst) == \old(strlen(dst)) + minimum(size, strlen(src));
```

These mistakes once again show that the specifications themselves can contain bugs and that the use of a tool to prove the specifications along with the code is essential.

4.3.12.3 Proof Results

The following are the results of running Z3:

```
b0: 5/5, b1: 7/7, b2: 19/21, default: 142/143, safety: 51/52
```

```
size_t
strlcat(char *dst, const char *src, size_t siz)
{
  char *d = dst;
  const char *s = src;
  size_t n = siz;
  size_t dlen;

  /* Find the end of dst and adjust bytes left but don't go past end */
  /*@
    loop assigns n, d;
    loop invariant 0 <= n <= siz;
    loop invariant \base_addr(d) == \base_addr(dst);
    loop invariant \valid(d);
    loop invariant 0 <= (d - dst) <= strlen{Pre}(dst);
    loop invariant d - dst == siz - n;
    loop invariant \forall integer k; 0 <= k < (d-dst) ==> dst[k] != '\0';
  */
  while (n-- != 0 && *d != '\0')
    d++;
  //@ assert siz < strlen{Pre}(dst) ==> (d-dst) == siz;
  //@ assert siz >= strlen{Pre}(dst) ==> (d-dst) == strlen{Pre}(dst);
  dlen = d - dst;
  //@ assert dlen <= siz;
  n = siz - dlen;
  if (n == 0)
    return(dlen + strlen(s));
  //@ assert n > 0;
  //@ assert siz > strlen{Pre}(dst);
  //@ assert (d-dst) == strlen{Pre}(dst);
  //@ assert n == (siz - strlen{Pre}(dst));

  //@ ghost char *p = d;
  //@ assert p-dst == strlen{Pre}(dst);
  //@ assert \forall integer k; 0 <= k < (d-p) ==> dst[k] == \at(dst[k], Pre);
  /*@
    loop assigns n, s, d, dst[p-dst..];
    loop invariant 1 <= n <= (siz - strlen{Pre}(dst));
    loop invariant \base_addr(d) == \base_addr(dst);
    loop invariant \base_addr(s) == \base_addr(src);
    loop invariant 0 <= (s-src) <= strlen(src);
    loop invariant \valid(s);
    loop invariant \valid(d);
    loop invariant n > 1 ==> d-p == s-src;
    loop invariant (siz - strlen{Pre}(dst)) - n == d - p;
    loop invariant \forall integer k; 0 <= k < (s-src) ==> src[k] != 0;
    loop invariant \forall integer k; 0 <= k <= strlen(src) ==> src[k] == \at(src[k], Pre);
    loop invariant \forall integer k; 0 <= k < (p-dst) ==> dst[k] == \at(dst[k], Pre);
    loop invariant \forall integer k; 0 <= k < strlen{Pre}(dst) ==> dst[k] == \at(dst[k], Pre);
    loop invariant \forall integer k; 0 <= k < (d-p) <= strlen(src)  ==>
      dst[k + strlen{Pre}(dst)] == src[k];
  */
  while (*s != '\0') {
    if (n != 1) {
      *d++ = *s;
      n--;
    }
    s++;
  }
  *d = '\0';
  //@ assert dst[d-dst] == 0;
  //@ assert d-dst == strlen(dst);
  //@ assert s-src == strlen(src);
  return(dlen + (s - src));    /* count does not include NUL */
}
```

Listing 16. *Strlcat function body and its contracts.*

4.3.12.4 Specification Details

The pre-conditions state that the function requires both strings to be valid and null-terminated strings. The destination buffer needs to be of length *siz*. It is also an important pre-condition that the destination buffer and the source string should not overlap.

The post-condition at line 7 states that the original string contents are preserved. The next post-condition at line 8 states that the function always returns the string it tried to create. These two post-conditions are common to all behaviours, therefore were put in the default behaviour.

Behaviour *b0* states that if *siz* is zero or the destination length is greater than the buffer size then there seems to be a mistake with the *siz* parameter so the function does not modify anything. The equal case covers the case when there is no room.

Behaviour *b1* states that if *siz* is greater than zero and there is room in the buffer for only the null character then the function only re-writes the null character so the buffer is modified but the length of the string is preserved. It is not clear why or if this is good behaviour for this function.

Behaviour *b2* states that if the size is not zero and there is room in the buffer for at least one character then the function copies elements from the source string. The number of characters copied depends on the following:

- If the length of the source string is less than the available space in the buffer (keeping 1 byte for the null character) then the function copies all of the source string and the length of the resultant string is the length of the original string plus the length of the source string.

- Otherwise, the function copies *siz - strlen{Old}(dst) - 1* characters from the source string up to the end of the buffer, appends a null character and the length of the resulting string is *siz-1*.

The first loop finds the end of the destination string without going past the end of the buffer. This means if the null character is not found before the end of the buffer, then the loop is stopped when *n* reaches zero. The assertions at lines 48 and 49 verify this.

The assertion at line 51 is necessary to prove the safety of the subtraction in terms of overflow as the result is assigned to an unsigned integer.

The assertions at lines from 55 to 58 help verify the intermediate state of the function after it does not return in previous code block.

The loop assigns at line 64 is important to prove proof obligations about the frame conditions of the function. The loop modifies the contents of *dst* by appending the source string starting from the index pointed by the ghost variable *p* that holds the end of the original string.

The loop invariant at line 65 expresses that for this loop, *n* is decremented from the value of *n* before the loop which is the remaining room left in the buffer.

The loop continues to iterate on the source string to find out its length for the return value even if the function stops copying so the invariant at line 71 captures that until *n* is equal to 1, pointer *d* and *s* are incremented the same amount of times.

The loop invariants below are important to prove the post-condition for the maximum number of elements copied:

```
loop invariant (siz - strlen{Pre}(dst)) - n == d - p;
loop invariant \forall integer k; 0 <= k < (d-p) <= strlen(src)  ==> dst[k + strlen{Pre}(dst)] == src[k];
```

The loop invariant at line 76 is important for proving proof obligations about the frame conditions of the function and expresses that the original elements of the buffer are not modified.

The assertions at the end of the function help prove the post-conditions for the length of resultant string. One of the proof obligations generated for the first assertion is the only one not proven for the default behaviour.

The missing safety proof obligation is for the following line:

```
  while (n-- != 0)
```

This is a legitimate concern as variable *n* is an unsigned integer so the post-decrement operation is not appropriate here. However, variable *n* is re-assigned immediately after the loop so it does not change the function’s behaviour.

5 Conclusions

Using static analysis and reasoning can significantly improve the quality of software and its documentation. This project was an attempt to use such techniques to reason about some of the string functions from the OpenBSD kernel. For each string function, specifications were written in ACSL and each function was analysed using Frama-C Framework to check their correctness with respect to the specifications written. The documentation was also examined in the light of the new specifications and the analysis performed.

Out of the twelve functions analysed, seven of them were fully proven and the rest had few missing proof obligations: two functions had one, one function had two and two functions had more. In total, 1208 proof obligations were automatically generated including 241 safety verification conditions that cover memory and integer safety. 1194 of these were proven and only 14 which constitute 1.2% of the total number were not proven.

No issues were detected in the code which is not surprising given the safety and reliability record of OpenBSD despite the pitfalls of C language in terms of security issues. Some minor issues and areas that possibly needed more clarification were highlighted for the documentation.

Using specifications and making documentation part of the source code with the support of static analysis tools as it was attempted in this project is a better approach than maintaining separate documentation. The specifications make more useful documentation than informal documentation such as comments or man pages. They can be checked automatically and therefore will be more precise and will be more likely to be kept consistent. Also, they are not ambiguous as natural language. Finally, as the ACSL specifications are part of structured comments, they do not affect the compilation of the code.

The specifications are also important for code maintenance as they record the intended behaviour of the code and can be checked automatically. Although string functions and their documentation are relatively less likely to be updated as often in terms of maintenance, it is important to prove their correctness as they are very prominent in the source tree and had often been prone to security issues in the past.

Most of the security issues encountered in the past for string functions were due to misuse. A good design should make it difficult for callers to misuse a function. This is also why it is very important that the documentation is as clear and precise as possible. However, even then, it does not help detecting issues automatically as in the case of specifications and static analysis. For example, with all the pre-conditions added to the string functions, it can be possible to catch many misuses and defects early and automatically in the code that calls these functions. As one of the criteria for choosing the functions for this project was utility, the benefits of static reasoning applied can naturally spread across the code base.

With all the benefits of specifications, there also is a cost for applying them. Writing specifications and proving them is hard and requires significant effort. Even for this project that used string functions which are relatively short and at reasonable complexity; it was hard enough to write and prove specifications for them.

It is even harder to write specifications for existing code as the experience of this project also shows. It would be easier if the code was written with specifications in mind from ground up. In some cases, complex or verbose specifications could have been avoided if the specifications were not added as an afterthought. Writing the specifications along with the code would also help verify the assumptions being made about the implementation and the design.

While the static reasoning helps detecting defects, it is also possible to make mistakes when writing the specifications themselves. The approach taken in this project where several versions of the specifications were written also demonstrated this. The tool support is crucial to verify specifications and catch mistakes. Writing the specifications solely based on the documentation is not enough without inspection of the code. First versions of the specifications were generally too weak and it was required to add loop invariants to strengthen the post-conditions. The second versions based on man files were generally close to the final version but mostly needed improvements and sometimes re-writing using behaviours to help track issues with proof.

Another mistake or pitfall, also experienced during this project, is unintentionally introducing inconsistencies in the specifications which cause some proof obligations to be erroneously proven. To catch these mistakes; a post-condition for proving *\false* was added to each function temporarily for test purposes. This is an important pitfall to watch for similar projects.

Finally, the tool, the Frama-C Framework, used for this project proved to be a good tool for the job. It was going through major releases at the time of writing and it promises to be even more powerful at its final release. Frama-C Framework is also well-documented and supported. One aspect worth mentioning was that writing the specifications using ACSL and the Jessie plug-in required many loop invariants to be written but it seems to be possible to use inference instead, although this was not explored for this project.

5.1 Future work

Further work to prove the missing verification conditions and checking the completeness of the specifications would be needed. In addition, only few of the string functions analysed call each other therefore it would also be beneficial to annotate and analyse the call sites. Also, adding more specifications to other areas of the kernel in a similar way would also be a good follow up to this project.

Frama-C Framework was evolving at the time of the writing. This meant that some ACSL features were not supported or were experimental; and there were some bugs not fixed in versions of Jessie plug-in and Why Platform used for this project. Further work could also attempt to use newer versions of Frama-C and Why Platform and re-write some of the specifications to use more and better ACSL constructs and possibly simplify some. Finally, by using the latest releases of the tools, the bug fix with Why Platform for functions *strcmp* and *strncmp* need to be verified.

Other aspects that can be explored further can be to separate the contracts into separate source files and the scripts used for the triage process. The scripts were really designed for the specific needs of the project. Although they were useful to inform the triage, such source code *grep'ing* cannot be very accurate. Their possible reuse would require some effort to extend and fix some bugs.

6 References

1. KindSoftware research group, http://kind.ucd.ie/

2. Ayewah, Pugh, Hovemeyer, Morgenthaler, Penix. Using Static Analysis to Find Bugs. IEEE Software. September/October 2008.

3. Frama-C Web Site, http://frama-c.cea.fr. Frama-C discussion list, http://frama-c-discuss@lists.gforge.inria.fr.

4. Hatcliff, Leavens, Leino, Müller, Parkinson. Behavioral Interface Specification Languages. CS-TR-09-01. March 2009.

5. Chalin, Kiniry, Leavens, Poll. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In Formal Methods for Components and Objects, Lecture Notes in Computer Science, vol. 4111. 2006.

6. Woodcock, Larsen, Bicarregui, Fitzgerald. Formal Methods: Practice and Experience. ACM Computing Surveys, Vol. 41, No. 4, Article 19. October 2009.

7. Kiniry, Chalin, Hurlin. Integrating Static Checking and Interactive Verification: Supporting Multiple Theories and Provers in Verification. In VSTTE. 2005.

8. Leavens, Leino, Poll, Ruby, Jacobs. JML: Notations and tools supporting detailed design in Java. Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications. 2000.

9. Kiniry, Fairmichael. Ensuring Consistency between Designs, Documentation, Formal Specifications, and Implementations. 12th International Symposium on Component-based Software Engineering (CBSE). 2009.

10. Meyer. Applying "Design by Contract". IEEE Computer. 1992.

11. Leavens, Cheon. Design by Contract with JML. http://www.jmlspecs.org/jmldbc.pdf. 2006.

12. OpenBSD Web Site, http://www.openbsd.org/

13. Why Platform, http://why.lri.fr/

14. Necula, McPeak, Rahul, Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. Proc. of Conference on Compiler Construction. 2002.

15. Moy. Automatic Modular Static Safety Checking for C Programs. PhD thesis. 2009.

16. Wikipedia, http://wikipedia.org

17. Evans and Larochelle. Improving Security Using Extensible Lightweight Static Analysis. IEEE SOFTWARE. January/February 2002.

18. Visual C++ Team Blog, http://blogs.msdn.com/vcblog/default.aspx

19. Evans. Splint Manual. Secure Programming Group, University of Virginia. 2003.

20. Baudin, Cuoq, Filliâtre, Marché, Monate, Moy, Prevosto. ACSL: ANSI/ISO C Specification Language. Version 1.4, Frama-C Beryllium implementation. http://frama-c.cea.fr. 2009.

21. MSDN, http://msdn.microsoft.com

22. Michael Howard's Web Log. http://blogs.msdn.com/michael_howard/default.aspx

23. Filliâtre, Marché. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In Computer Aided Verification (CAV). 2007.

24. Moy and Marché. Jessie Plugin Tutorial, Beryllium Version. http://frama-c.cea.fr. 2009.

25. Tools: Cygwin, www.cygwin.com. Gcc, http://gcc.gnu.org. Ocaml, http://caml.inria.fr. Trac, http://trac.edgewall.org. Strawberry Perl, http://strawberryperl.com/strawberry-perl-5.10.0.6.

26. Theorem provers: Simplify, http://kind.ucd.ie/products/opensource/Simplify/. Alt-Ergo, http://alt-ergo.lri.fr/. Z3, http://research.microsoft.com/en-us/um/redmond/projects/z3/

27. Contracts in OpenBSD Project Trac Site, http://trac.ucd.ie/trac.

28. Cok, Kiniry, Poll. Specification Tips and Pitfalls. ESC/Java2 & JML Tutorial.

29. Miller, de Raadt. Strlcpy and Strlcat— Consistent, Safe, String Copy and Concatenation. In Proceedings of the USENIX Technical Conference, Freenix Track. 1999.