# Context Generation from Formal Specifications for C Analysis Tools

Michele Alberti[1]* and Julien Signoles[2]

[1] TrustInSoft, Paris, France
michele.alberti@trust-in-soft.com
[2] CEA LIST, Software Reliability and Security Laboratory
F-91191 Gif-sur-Yvette Cedex, France
julien.signoles@cea.fr

**Abstract.** Analysis tools like abstract interpreters, symbolic execution tools and testing tools usually require a proper context to give useful results when analyzing a particular function. Such a context initializes the function parameters and global variables to comply with function requirements. However it may be error-prone to write it by hand: the handwritten context might contain bugs or not match the intended specification. A more robust approach is to specify the context in a dedicated specification language, and hold the analysis tools to support it properly. This may mean to put significant development efforts for enhancing the tools, something that is often not feasible if ever possible.
This paper presents a way to systematically generate such a context from a formal specification of a C function. This is applied to a subset of the ACSL specification language in order to generate suitable contexts for the abstract interpretation-based value analysis plug-ins of Frama-C, a framework for analysis of code written in C. The idea here presented has been implemented in a new Frama-C plug-in which is currently in use in an operational industrial setting.

**Keywords:** Formal Specification, Code Generation, Transformation, Code Analysis, Frama-C, ACSL

## 1 Introduction

Code analysis tools are nowadays effective enough to be able to provide suitable results on real-world code. Nevertheless several of these tools including abstract interpreters, symbolic execution tools, and testing tools must analyze the whole application from the program entry point (the *main* function); or else either they just cannot be executed, or they provide too imprecise results. Unfortunately such an entry point does not necessarily exist, particularly when analyzing libraries.

In such a case, the verification engineer must manually write the context of the analyzed function $f$ as a main function which initializes the parameters of $f$ as well as the necessary global variables. This mandatory initialization step must enforce the function requirements and may restrict the possible input values for the sake of memory footprint and time efficiency of the analysis. This approach is however error-prone:

* This work was done when the first author was at CEA LIST, Software Reliability and Security Laboratory.

additionally to usual pitfalls of software development (*e.g.* bugs, code maintenance, *etc.*), the handwritten context may not match the function requirements, or be over restrictive. Moreover this kind of shortcomings may be difficult to detect due to the fact that the context is not explicitly the verification objective.

A valid and more robust alternative is to specify such a context in a dedicated specification language, and make the analysis tools handle it properly. This is often an arduous approach as the support for a particular specification language feature may entail a significant development process, something that is often not feasible if ever possible. Also, it requires to do so for every tool.

This paper presents a way to systematically generate an analysis context from a formal specification of a C function. The function requirements as well as the additional restrictions over the input domains are expressed as function preconditions in the ANSI/ISO C Specification Language (in short, ACSL) [2]. This specification $\mathcal{S}$ is interpreted as a constraint system, simplified as much as possible, then converted into a C code $\mathcal{C}$ which exactly implements the specification $\mathcal{S}$. Indeed not only every possible execution of $\mathcal{C}$ satisfies $\mathcal{S}$ but conversely, there is an execution of $\mathcal{C}$ for every possible input satisfying the constraints expressed by $\mathcal{S}$. We present the formalization of this idea for an expressive subset of ACSL including standard logic operators, integer arithmetic, arrays and pointers, pointer arithmetic, and built-in predicates for the validity and initialization properties of memory location ranges.

We also provide implementation details about our tool, named CfP for *Context from Preconditions*, implemented as a Frama-C plug-in. Frama-C is a code analysis framework for code written in C [11]. Thanks to the aforementioned technique, CfP generates suitable contexts for two abstract interpretation-based value analysis tools, namely the the Frama-C plug-in EVA [3] and TIS-Analyzer [8] from the TrustInSoft company. Both tools are actually distinct evolved versions of an older plug-in called Value [6]. In particular, TrustInSoft successfully used CfP on the mbed-TLS library (also known as PolarSSL), an open source implementation of SSL/TLS[3], when building its verification kit [21]. It is worth noting that CfP revealed some mistakes in contexts previously written by hand by expert verification engineers when comparing its results with these pieces of code. Also, CfP generates code as close as possible to human-written code: it is quite readable and follows code patterns that experts of these tools manually write.

*Contributions* The contributions of this paper are threefold: **a novel technique to** systematically **generate an analysis context** from a formal specification of a C function, **a precise formalization** of this technique, and a presentation of **a tool** implementing this technique which is **used in an operational industrial setting**.

*Outline* Section 2 presents an overview of our technique through a motivating example. Section 3 details preconditions to constraints conversion, while Section 4 explains the C code generation scheme for these latter. Section 5 evaluates our approach and Section 6 discusses related work. Section 7 concludes this work by also discussing future work.

[3] `https://tls.mbed.org/`

## 2 Overview and Motivating Example

We illustrate our approach on context generation through the function `aes_crypt_cbc`, a cryptographic utility implemented by the mbed-TLS library. Figure 1 shows its prototype and ACSL preconditions as written by TrustInSoft for its verification kit [21].

```
1  typedef struct {
2    int nr;                     /*  number of rounds  */
3    unsigned long *rk;          /*  AES round keys    */
4    unsigned long buf[68];      /*  unaligned data    */
5  } aes_context;
6
7  /*@ requires ctx_valid: \valid(ctx);
8    @ requires ctx_init: \initialized(ctx->buf + (0 .. 63));
9    @ requires ctx_rk: ctx->rk == ctx->buf;
10   @ requires ctx_nr: ctx->nr == 14;
11   @ requires mode: mode == 0 || mode == 1;
12   @ requires length: 16 <= length <= 16672;
13   @ requires length_mod: length % 16 == 0;
14   @ requires iv_valid: \valid(iv + (0 .. 15));
15   @ requires iv_init: \initialized(iv + (0 .. 15));
16   @ requires input_valid: \valid_read(input + (0 .. length - 1));
17   @ requires input_init: \initialized(input + (0 .. length - 1));
18   @ requires output_valid: \valid(output + (0 .. length - 1)); */
19 int aes_crypt_cbc(aes_context *ctx,int mode,size_t length,unsigned char iv[16],
20                   const unsigned char *input,unsigned char *output);
```

**Fig. 1:** ACSL preconditions of the mbed-TLS function `aes_crypt_cbc`.

*Specification* The function `aes_crypt_cbc` provides encryption and decryption of a buffer according to the AES cryptographic standard and the CBC encryption mode. The function takes six parameters. The last two are the input and the output strings. The parameter `ctx` stores the necessary information to the AES substitution-permutation network, in particular the number of rounds and the round keys defined in a dedicated structure at lines 1–5. The parameter `mode` indicates whether the function should encrypt or decrypt the `input`. The parameter `length` indicates the length of the `input` string. Finally the parameter `iv` provides an initialization vector for the `output` of 16 characters (`unsigned char iv[16]`). This declared length is actually meaningless for most C tools because an array typed parameter is adjusted to have a pointer type [10, Section 6.9.1 and also footnote 79 at page 71], but CfP nevertheless considers it as part of the specification in order to generate a more precise context.

ACSL annotations are enclosed in `/*@ ... */` as a special kind of comments. Therefore they are ignored by any C compiler. A function precondition is introduced by the keyword **`requires`** right before the function declaration or definition. It must be satisfied at every call site of the given function. Here the function `aes_crypt_cbc` has 12 precondition clauses, and the whole function precondition is the conjunction of all of them. Clauses may be tagged with names, which are logically meaningless but provide a way to easily refer to and to document specifications. For instance, the first precondition (line 7) is named `ctx_valid` while the second (line 8) is named `ctx_init`.

We now detail the meaning of each precondition clause. All pointers must be valid, that is properly allocated, and point to a memory block of appropriate length that the program can safely access either in read-only mode (predicate **`\valid_read`**), or in read-write mode (predicate **`\valid`**) . That is the purpose of preconditions `ctx_valid`,

`iv_valid`, `input_valid` and `output_valid`: `ctx` must point to a memory block containing at least a single `aes_context` struct, `iv` must be able to contain at least 16 unsigned characters (ranging from 0 to 15), while `input` and `output` must be able to contain at least `length` unsigned characters (ranging from 0 to `length` − 1). Memory locations, which are read by the function, must be properly initialized. That is the purpose of the precondition clauses `ctx_init`, `iv_init`, and `input_init` which initialize the first 64 cells of `ctx->buf` as well as every valid cell of `iv` and `input`. The specification clause `mode` specifies that the mode must be either 0 (encryption) or 1 (decryption), while the specification clause `length_mod` specifies that the length should be a multiple of the block size (*i.e.* 16) as specified in mbed-TLS. The other clauses restrict the perimeter of the analysis in order to make it tractable.

The clause `ctx_rk` is a standard equality for an AES context, while the clause `ctx_nr` is true for 256-bit encryption keys. Finally the clause `length` aims to restrict the analysis to buffers of size from 16 to 16672 unsigned characters.

*Context Generation* A naive approach for context generation would consider one precondition clause after the other and directly implement it in C code. However, this would not work, in general, since requirements cannot be treated in any order. In our running example, for instance, variables `input` and `output` depends on the variable `length`: the precondition clauses over this latter must be treated before those over the former, as well as the generated code for these variables must initialize the latter, first, and the former afterwards, to be sound. To solve such problems, one could first record every dependency among the left-values involved in the specification, and then proceed to generate C code accordingly. An approach based only on a dependency graph is nonetheless insufficient for those preconditions that need an inference reasoning in order to be implemented correctly. As an example, treating the precondition `/*@requires \valid(x+(0..3)) && *(x+4)==1;*/` demands to infer `x` as an array of 5 elements in order to consider the initialization `x[4] = 1;` correct.

We now give an overview on how we treat context generation by means of the plug-in CfP of Frama-C. On the `aes_crypt_cbc` function contract, CfP provides the result shown in Figure 2 (assuming that the size of `unsigned long` is 4 bytes[4]).

First note that every execution path ends by a call to the function `aes_crypt_cbc`. Up to these calls, the code initializes the context variables (prefixed by `cfp`) in order to satisfy the precondition of this function, while the different paths contribute to cover all the cases of the specification. The initialization code is generated from sets of constraints that are first inferred for every left-value involved in the precondition. While inferring these constraints from the precondition clauses, the implicit dependencies among left-values are made explicit and recorded in a dependency graph. This latter is finally visited to guide the code generation process in order to obtain correct C code.

Let us start detailing the generated code for both preconditions about `length` (Figure 1, lines 12–13). First CfP declares a variable `cfp_length` of the same type as `length` (line 4). Then it initializes it by means of the Frama-C library function `Frama_C_unsigned_int_interval` (line 7). It takes two `unsigned int` arguments and returns a random value comprised between the two. This allows to fulfill

[4] This kind of system-dependent information is customizable within Frama-C.

```
1  int cfp_aes_crypt_cbc(void) {
2    unsigned char *cfp_output, *cfp_input;
3    unsigned char cfp_iv[16];
4    size_t cfp_length;
5    aes_context cfp_ctx;
6    int cfp_disjunction;
7    cfp_length = Frama_C_unsigned_int_interval(16, 16672);
8    if (cfp_length % 16 == 0) {
9      Frama_C_make_unknown((char *)cfp_ctx.buf,256);
10     cfp_ctx.nr = 14;
11     cfp_ctx.rk = cfp_ctx.buf;
12     Frama_C_make_unknown((char *)cfp_iv,16);
13     cfp_input = (unsigned char *)malloc(cfp_length);
14     if (cfp_input != (unsigned char *)0) {
15       Frama_C_make_unknown((char *)cfp_input, cfp_length);
16       cfp_output = (unsigned char *)malloc(cfp_length);
17       if (cfp_output != 0) {
18         cfp_disjunction = Frama_C_int_interval(0,1);
19         if (cfp_disjunction) {
20           int cfp_mode;
21           cfp_mode = 1;
22           aes_crypt_cbc(&cfp_ctx,cfp_mode,cfp_length,cfp_iv,cfp_input,cfp_output);
23         }
24         else {
25           int cfp_mode;
26           cfp_mode = 0;
27           aes_crypt_cbc(&cfp_ctx,cfp_mode,cfp_length,cfp_iv,cfp_input,cfp_output);
28         }
29       }
30     }
31   }
32   return 0;
33 }
```

**Fig. 2:** Slightly simplified version of the code generated by CfP for the specification in Figure 1. Compared to the actual version, only a few integer casts have been removed for reasons of brevity.

the former requirement and to guarantee that Frama-C-based abstract interpreters will interpret this result with exactly the required interval. Also, it corresponds to the way that expert engineers would write a general context for such analyzers. Finally, the requirement `length % 16 == 0` is implemented by the conditional at line 8.

Lines 9–11 implement the preconditions about `ctx`, a pointer to an `aes_context`. Instead of allocating such a pointer, the generated code just declares a local variable `cfp_ctx` and passes its address to the function calls. This automatically satisfies the precondition on pointer validity. Line 9 initializes the 256 first bytes of the structure field `buf` by using the Frama-C library function `Frama_C_make_unknown`. Assuming that the size of `unsigned long` is 4 bytes, 256 bytes is the size of 64 values of type `unsigned long`. Again, an expert engineer would also use this library function. Lines 10 and 11 initialize the fields `ctx->nr` and `ctx->rk` by single assignments. Here CfP fulfills the equality requirement `ctx->rk == ctx->buf` with respect to `ctx->rk` instead of `ctx->buf` because the latter already refers to a memory buffer.

The requirements on function arguments `iv`, `input`, and `output` are implemented by lines 12–17. Let us just point out how CfP defines the respective variables: while `ctx_iv` is as an array of 16 `unsigned char`, `ctx_input` and `ctx_output` are just pointers to dynamically allocated memory buffers. Indeed, while CfP can infer the exact dimension of the former from the specification, the dimension of these latter depends on the value of `ctx_length`, which is determined only at runtime.

The last part of the generated code (lines 18–29) handles the requirement on `mode`, which is either 0 or 1. Although the generated conditional may seem excessive in the case of these particular values, it is nonetheless required in the general case (for instance, consider the formula `mode == 5 || mode == 7`).

## 3 Simplifying ACSL Preconditions into State Constraints

This section presents a way to systematically reduce a function precondition to a set of constraints on the function context (*i.e.* function parameters and global variables).

We first introduce an ACSL-inspired specification language on which we shall formalize our solution. Then, we define the notion of state constraint as a form of requirement over a C left-value, which in turn we generate as C code for initializing it. In order to simplify state constraints the most, we make use of symbolic ranges, originally introduced by Blume and Eigenmann [4] for compiler optimization. We finally provide a system of inference rules that formalizes such a simplification process.

### 3.1 Core Specification Language

In this work we shall consider the specification language in Figure 3. It is almost a subset of ACSL [2] but for the predicate `defined`, which subsumes the ACSL predicates **`\initialized`** and **`\valid`** (see below).

| | | | |
|---|---|---|---|
| Predicates | $P ::=$ | $T\ \texttt{cop}\ T$ | term comparison ($\texttt{cop} \in \{\equiv, \leq, <, \geq, >\}$) |
| | | $\texttt{defined}(M)$ | $M$ is defined |
| | | $P \wedge P \mid P \vee P \mid \neg P$ | logic formula |
| Terms | $T ::=$ | $z$ | integer constant ($z \in \mathbb{Z}$) |
| | | $M$ | memory value |
| | | $T\ \texttt{bop}\ T$ | arithmetic operation ($\texttt{bop} \in \{+, -, \times, /, \%\}$) |
| Memory Values | $M ::=$ | $L$ | left-value |
| | | $M$ ++ $T$ | single displacement |
| | | $M$ ++ $T$ . . $T$ | displacement range |
| Left-Values | $L ::=$ | $x$ | C variable |
| | | $\star M$ | dereference |
| Types | $\kappa ::=$ | $\iota$ | integer |
| | | $\kappa\star$ | pointer |

**Fig. 3:** Predicates, terms, and types.

Predicates are logic formulæ defined on top of typed term comparisons and predicates `defined`. Terms are arithmetic expressions combining integer constants and memory values by means of the classic arithmetic operators. Memory values include left-values, which are C variables and pointer dereferences ($\star$), and memory displacements through the operator (++). In particular, $M$ ++ $T_1$ . . $T_2$ defines the set of memory values $\{M \text{ ++ } T_1, \ldots, M \text{ ++ } T_2\}$ and may only appear as the outermost construct in a predicate `defined`. On integers, $\texttt{defined}(L)$ holds whenever $L$ is an initialized left-value. On pointers, $\texttt{defined}(M)$ holds whenever $M$ is a properly allocated and initialized memory region.

*Term typing* Terms of our language are typed. A left-value may take either an integer ($\iota$) or a pointer ($\kappa\star$) type, while memory values are pointers. We omit the typing rules for terms, which are quite standard. Let us just specify that memory values of the form $M$ ++ $T$ have pointer type, as well as the recursive occurrence $M$, while $T$ must have integer type. (Memory values $M$ ++ $T$ . . $T$ are typed as set of pointers [2].) Since we do not consider any kind of coercion construct, terms of pointer type cannot appear where integer terms are expected, that is, they cannot appear in arithmetic expressions. It also follows that term comparisons only relate terms of the same type.

*Term normal forms* For the sake of concision and simplicity, the remainder of this work assumes some simplifications to take place on terms in order to consider term normal forms only. In particular, arithmetic expressions are maximally flattened and factorized (*e.g.* by means of constant folding techniques, *etc.*). We will conveniently write single displacements $M$ ++ $T$ as $M$ ++ $T$ . . $T$. We also assume memory values with displacement ranges to be either of the form $x$ ++ $T_1$ . . $T_2$ or $\star L$ ++ $T_1$ . . $T_2$. To this end, terms of the form ($L$ ++ $T_1$ . . $T_2$) ++ $T_3$ . . $T_4$ simplify into $L$ ++ ($T_1$ + $T_3$) . . ($T_2$ + $T_4$). Finally, memory values $L$ ++ 0 . . 0 normalize to $L$.

*Disjunctive normal forms* A precondition is a conjunction of predicate clauses, each one given by an ACSL `requires` (*cf.* example in Figure 1). As a preliminary step, we shall rewrite this conjunctive clause into its disjunctive normal form $\bigvee_i \bigwedge_j P_{ij}$, where each $P_{ij}$ is a *predicate literal* (or simply *literal*), that is, a predicate without nested logic formulæ. A *negative literal* is either of the form $\neg\texttt{defined}(M)$ or $\neg(M_1 \equiv M_2)$, with $M_1, M_2$ pointers, as every other negative literal in the input predicates is translated into a positive literal by applying standard arithmetic and logical laws. A non-negative literal is called a *positive literal*. Most of the rest of this section focuses on positive literals: negative literals and conjunctive clauses are handled in the very end, while disjunctive clauses will be considered when discussing code generation in Section 4.

### 3.2 State Constraints

We are interested in simplifying a predicate literal into a set of constraints over C left-values, called *state constraints*. These are meant to indicate the minimal requirements that the resulting C function context must implement for satisfying the function precondition. In Section 4, they will be, in turn, converted into C code.

We intuitively consider a state constraint to represent the domain of definition of a C left-value of the resulting function context state. Since such domains might not be determined in terms of integer constants only, we shall found their definition on the notion of symbolic ranges [4]. As we want to simplify state constraints the most, we define them in terms of the symbolic range algebra proposed by Nazaré et al. [14]. Our definitions are nonetheless significantly different, even though inspired from their work.

*Symbolic Expressions* A *symbolic expression* $E$ is defined by the following grammar, where $z \in \mathbb{Z}$, $\texttt{bop} \in \{\texttt{+}, \texttt{-}, \times, \texttt{/}, \texttt{\%}\}$, and `max` and `min` are, respectively, the largest and the smallest expression operators. We denote $\mathbb{E}$ the set of symbolic expressions.

$$E ::= z \mid x \mid \star E \mid E\ \texttt{bop}\ E \mid \texttt{max}(E, E) \mid \texttt{min}(E, E).$$

In the rest of this section, we assume a mapping from memory values to their respective symbolic expression, and let the context discriminate the former from the latter.

In Section 3.3 we shall simplify symbolic expressions. For this, we need a domain structure. Let us denote $\mathbb{E}_\infty = \mathbb{E} \cup \{-\infty; +\infty\}$ and $\mathbb{Z}_\infty = \mathbb{Z} \cup \{-\infty; +\infty\}$. We define a *valuation of a symbolic expression* $E$ every map $\mathcal{V}(E)$, from $\mathbb{E}_\infty$ to $\mathbb{Z}_\infty$, obtained by substituting every C variable in $E$ with a distinct integer, the symbol $\star$ with a natural number strictly greater than 1 as a multiplicative coefficient, and interpreting the operators $\{\texttt{bop}, \texttt{min}, \texttt{max}\}$ as their respective functions over $\mathbb{Z}_\infty \times \mathbb{Z}_\infty$. If we denote $\leq_\infty$ the standard ordering relation on $\mathbb{Z}_\infty$, then the preorder $\preccurlyeq$ on $\mathbb{E}_\infty$ is defined as follows:

$$E_1 \preccurlyeq E_2 \iff \forall \mathcal{V}, \mathcal{V}(E_1) \leq_\infty \mathcal{V}(E_2).$$

The partial order $\preceq$ over $\mathbb{E}_\infty$ is therefore the one induced from $\preccurlyeq$ by merging in the same equivalence class elements $x$ and $y$ of $\mathbb{E}_\infty$ such that $x \preccurlyeq y$ and $y \preccurlyeq x$. As an example, the elements $0$ and $\texttt{min}(0, 0)$ are equivalent.

*Lattice of Symbolic Expression Ranges* A *symbolic range* $R$ is a pair of symbolic expressions $E_1$ and $E_2$, denoted $[E_1, E_2]$. Otherwise said, a symbolic range is an interval with no guarantee that $E_1 \preceq E_2$. We denote $\mathbb{R}$ the set of symbolic ranges extended with the empty range $\emptyset$ and $\sqsubseteq$ its partial ordering which is the usual partial order over (possibly empty) ranges. Any symbolic range $[E_1, E_2]$ such that $E_2 \prec E_1$ is therefore equivalent to $\emptyset$. Consequently $(\mathbb{R}, \sqsubseteq)$ is a domain. Its infimum is $\emptyset$ while its supremum is $[-\infty, +\infty]$. We denote $\sqcup$ and $\sqcap$ its join and meet operators, respectively. It is worth noting that, given $(E_i)_{1 \leq i \leq 4}$ four symbolic expressions, the following equations hold:

$$[E_1, E_2] \sqcup [E_3, E_4] = [\texttt{min}(E_1, E_3), \texttt{max}(E_2, E_4)]$$
$$[E_1, E_2] \sqcap [E_3, E_4] = [\texttt{max}(E_1, E_3), \texttt{min}(E_2, E_4)]\,.$$

In words, `min` and `max` are compliant with our ordering relations. In Section 3.3, when simplifying literals, they will be introduced as soon as incomparable formulæ will be associated to the same left-value, resulting into an unsimplifiable constraint. Also, it is worth noting that $\sqcup$ and $\sqcap$ are, in general, not statically computable operators. To solve this practical issue, when these are not computable on some symbolic expressions, CfP relies on the above equations in order to delay their evaluations at runtime. Eventually, the code generator will convert them into conditionals.

*State Constraints as Symbolic Ranges with Runtime Checks* Symbolic ranges capture most minimal requirements over the C left-values of a function precondition: for integer typed left-values, a symbolic range represents the integer variation domain, while for pointer typed left-values, it represents a region of valid offsets. They are commonly used in abstract interpreters for range [7,13] and region analysis [14,18], respectively.

However, some predicate literals cannot be simplified into symbolic ranges, requiring their encoding as *runtime checks*, that is, to be verified at runtime by means of conditionals. We denote $\texttt{RTC}(T_1\ \texttt{cop}\ T_2)$ a runtime check between two terms $T_1$ and $T_2$. We then call *state constraint* any pair $C = R \oplus X$ given by a symbolic range $R$ and a set $X$ of runtime checks. We denote $\pi_1(C)$ (resp. $\pi_2(C)$) the first (resp. the second) projection of $C$, that is, $R$ (resp. $X$).

### 3.3 Inferring State Constraints

We now formalize our solution for simplifying a positive literal into a set of state constraints as a system of inference rules. Negative literals, as well as conjunctive clauses, are handled separately at the end of the section.

*Simplification Judgments* Simplification rules are given over judgments of the form

$$\Sigma \vdash P \Rightarrow \Sigma',$$

where $P$ is a predicate literal, and $\Sigma$, $\Sigma'$ are maps from left-values to state constraints. Each judgment associates a set of state constraints $\Sigma$ and a literal $P$ with the result of simplifying $P$ with respect to the left-values appearing in it, that is, an updated map $\Sigma'$ equal to $\Sigma$ but for the state constraints on these latter. Figures 4 shows the formalization of the main literal simplifications. This system does not assume the consistency of the precondition: if this is inconsistent, no rule applies and the simplification process fails.

*Predicates* `defined` Figure 4a provides the simplification rules for literal `defined`. Rules VARIABLE and DEREFERENCE enforce the initialization of a left-value $L$ in terms of the symbolic range $neutral\_ival(\kappa)$. This latter is respectively defined as $\emptyset$, for $\kappa$ a pointer type, and $[-\infty, +\infty]$, for $\kappa$ integer type. These are quite common initial approximations when inferring variation domains of either memory or integer values.

Rules RANGE-1 and RANGE-2 enforce the validity of a memory region determined by the displacement range $L$ ++ $(T_1 \,..\, T_2)$. The first premise of these rules established whether $L$ is already enforced in $\Sigma$ to be an alias of a memory value $M$, as indicated by the singleton range $[M;\, M]$. If not, rule RANGE-1 first enforces the initialization of $L$ and the soundness of the displacement bound determined by $T_1$ and $T_2$, and then it updates the region of valid offsets pointed to by $L$ to include the range $[0;\, T_2]$. In practice, predicates $0 \leq T_1 \leq T_2$ are added only if not statically provable. Moreover, note that we do not consider $T_1$ as the lower bound of the symbolic range, because C memory regions must start at index 0. Rule RANGE-2 handles the case of $L$ alias of $M$ in $\Sigma$ by enforcing the validity of the memory region determined by $M$ to take into account the displacement range $(T_1 \,..\, T_2)$. In particular, since single displacements only may appear in memory equality predicates (*cf.* rule MEMORY-EQ), $M$ is of the form $L'$ ++ $(T_3 \,..\, T_3)$, and the validity of the alias $L$ within the range $(T_1 \,..\, T_2)$ is obtained by requiring the validity of the displacement range $L'$ ++ $(\texttt{min}(T_1, T_3) \,..\, \texttt{max}(T_2, T_3))$.

Rule IDEMPOTENCE is provided only to allow the inference process to progress.

*Term comparison predicates* Rules in Figure 4b formalize the simplification of integer term comparison and memory equality predicates. The first two are actually rule schema, as CMP-1 and CMP-2 describe term comparison simplifications over the integer comparison operators $\{\equiv, \leq, \geq\}$. (Strict operators are treated in terms of non-strict ones.) Let us detail rule CMP-1 with respect to a generic operator `cop`. The rule applies whenever $T_1$ `cop` $T_2$ can be rewritten by means of classic integer arithmetic transformations as $L$ `cop` $T_3$, that is, as a left-value in relation `cop` with an integer term $T_3$. If so, CMP-1 reduces the symbolic range of $L$ with respect to the one given by $ival(\texttt{cop}, T_3)$. This latter function takes a comparison operator `cop` and an integer

IDEMPOTENCE
$$\frac{L \in \Sigma}{\Sigma \vdash \texttt{defined}(L) \Rightarrow \Sigma}$$

VARIABLE
$$\frac{x \notin \Sigma \qquad type(x) = \kappa \qquad \Sigma' = \Sigma \cup \{x \mapsto neutral\_ival(\kappa)\}}{\Sigma \vdash \texttt{defined}(x) \Rightarrow \Sigma'}$$

DEREFERENCE
$$\frac{\begin{array}{c}\star M \notin \Sigma \qquad \Sigma \vdash \texttt{defined}(M) \Rightarrow \Sigma' \\ type(\star M) = \kappa \qquad \Sigma'' = \Sigma' \cup \{\star M \mapsto neutral\_ival(\kappa)\}\end{array}}{\Sigma \vdash \texttt{defined}(\star M) \Rightarrow \Sigma''}$$

RANGE-1
$$\frac{\begin{array}{c}\pi_1(\Sigma\,(L)) \neq [M;\, M] \\ \Sigma \vdash \texttt{defined}(L) \wedge 0 \leq T_1 \leq T_2 \Rightarrow \Sigma' \qquad \Sigma'' = \Sigma' \left[L \leftarrow \pi_1(\Sigma'\,(L)) \sqcup [0;\, T_2]\right]\end{array}}{\Sigma \vdash \texttt{defined}(L \mathbin{+\!\!+} (T_1 \,..\, T_2)) \Rightarrow \Sigma''}$$

RANGE-2
$$\frac{\begin{array}{c}\pi_1(\Sigma\,(L)) = [M;\, M] \qquad \text{base}(M) = L' \qquad \text{offset}(M) = T_3 \\ \Sigma \vdash \texttt{defined}(L' \mathbin{+\!\!+} (\texttt{min}(T_1, T_3) \,..\, \texttt{max}(T_2, T_3))) \Rightarrow \Sigma'\end{array}}{\Sigma \vdash \texttt{defined}(L \mathbin{+\!\!+} (T_1 \,..\, T_2)) \Rightarrow \Sigma'}$$

**(a)** Simplification of literal `defined`.

CMP-1
$$\frac{\begin{array}{c}L \in \{T_1, T_2\} \\ T_1 \mathbin{\texttt{cop}} T_2 \rightsquigarrow L \mathbin{\texttt{cop}} T_3 \qquad \Sigma \vdash \texttt{defined}(L) \wedge \bigwedge_{L' \in T_3} \texttt{defined}(L') \Rightarrow \Sigma' \\ \Sigma'' = \Sigma' \left[L \leftarrow \pi_1(\Sigma'\,(L)) \sqcap ival(\texttt{cop}, T_3)\right]\end{array}}{\Sigma \vdash T_1 \mathbin{\texttt{cop}} T_2 \Rightarrow \Sigma''}$$

CMP-2
$$\frac{\begin{array}{c}\Sigma \vdash \bigwedge_{L \in \{T_1, T_2\}} \texttt{defined}(L) \Rightarrow \Sigma' \\ L \in \{T_1, T_2\} \qquad \Sigma'' = \Sigma' \left[L \leftarrow \pi_2(\Sigma'\,(L)) \cup \texttt{RTC}(T_1 \mathbin{\texttt{cop}} T_2)\right]\end{array}}{\Sigma \vdash T_1 \mathbin{\texttt{cop}} T_2 \Rightarrow \Sigma''}$$

MEMORY-EQ
$$\frac{\begin{array}{c}i, j \in \{1, 2\} \wedge i \neq j \\ \text{base}(M_{\{i,j\}}) = L_{\{i,j\}} \qquad \text{offset}(M_{\{i,j\}}) = T_{\{i,j\}} \qquad T_3 = T_j + (-T_i) \\ M' = L_j \mathbin{+\!\!+} (T_3 \,..\, T_3) \qquad \Sigma \vdash \texttt{defined}(L_i) \wedge \texttt{defined}(M') \Rightarrow \Sigma' \\ \pi_1(\Sigma'\,(L_i)) \sqsubseteq \pi_1(\Sigma'\,(L_j)) \qquad \Sigma'' = \Sigma' \left[L_i \leftarrow \left[M';\, M'\right]\right]\end{array}}{\Sigma \vdash M_1 \equiv M_2 \Rightarrow \Sigma''}$$

**(b)** Simplification of term comparison and memory equality literals.

NOT-DEFINED
$$\frac{M \notin \Sigma}{\Sigma \vdash \neg\, \texttt{defined}(M) \Rightarrow \Sigma}$$

MEMORY-NEQ
$$\frac{\begin{array}{c}\Sigma \vdash \texttt{defined}(M_1) \wedge \texttt{defined}(M_2) \Rightarrow \Sigma' \qquad i, j \in \{1, 2\} \wedge i \neq j \\ \text{base}(M_{\{i,j\}}) = L_{\{i,j\}} \qquad [L_i;\, L_i] \not\sqsubseteq \pi_1(\Sigma'\,(L_j)) \qquad [L_j;\, L_j] \not\sqsubseteq \pi_1(\Sigma'\,(L_i))\end{array}}{\Sigma \vdash M_1 \not\equiv M_2 \Rightarrow \Sigma'}$$

**(c)** Simplification of negative literals.

**Fig. 4:** Simplification of literals into state constraints.

term $T$ as arguments, and returns as result the symbolic range $[T; T]$ when `cop` is $\equiv$, $[-\infty; T]$ (resp. $[T; +\infty]$) when `cop` is $\leq$ (resp. $\geq$). Since both $L$ and $T_3$ are integer typed terms, there is no aliasing issue here. Rule CMP-2 can always be applied, although we normally consider it when CMP-1 cannot. In that case, rule CMP-2 conservatively enforces the validity of the term comparison by means of a runtime check.

*Aliasing* Rule MEMORY-EQ handles aliasing between two pointers with single displacement $M_1$ and $M_2$. Assuming both of the form $L_{\{i,j\}}$ ++ $T_{\{i,j\}}$, with distinct $i, j \in \{1, 2\}$, a pointer $M'$ is first defined as $L_j$ with single displacement $T_3$, this latter determined by summing the offsets $-T_i$ and $T_j$ together. Such a pointer is then enforced to be defined, and in the case that the actual region pointed by $L_j$ is established to be larger then the one pointed by $L_i$, then $L_i$ is considered an alias of $M'$. Although rather conservative, due to the fact that $\sqsubseteq$ is not statically computable in general, the second to last premise is important for ensuring soundness.

*Negative literals* Figure 4c shows the rules for negative literals. These rules do not simplify literals into state constraints, but rather ensure precondition consistency. For instance, $\neg\texttt{defined}(x) \wedge x == 0$ is inconsistent as $x$ should be defined with value 0 and undefined at the same time. In such a case, the system must prevent code generation.

Rule NOT-DEFINED just checks that the memory value $M$ does not appear in the map $\Sigma$, which suffices to ensure that $M$ is not yet defined.

Rule MEMORY-NEQ applies under the hypothesis that both pointers $M_1$ and $M_2$ determine different memory regions. In particular, the two are not aliases whenever each base address of one pointer does not overlap with the memory region of the other.

*Conjunctive Clauses* $\bigwedge_i P_i$, on either positive or negative literals $P_i$, are handled sequentially through the following AND rule. Given the definition of MEMORY-NEQ and NOT-DEFINED, it assumes that negative literals are treated only after the positive ones, by exhaustively applying rule MEMORY-NEQ first, and rule NOT-DEFINED afterwards.

$$\textsc{And}\quad \frac{\Sigma_0 \vdash P_1 \Rightarrow \Sigma_1 \qquad \Sigma_1 \vdash P_2 \Rightarrow \Sigma_2 \qquad \cdots \qquad \Sigma_{n-1} \vdash P_n \Rightarrow \Sigma_n}{\Sigma_0 \vdash \bigwedge_i P_i \Rightarrow \Sigma_n}$$

*Dependency Graph on Memory Values* On a conjunctive clause, the system of inference rules in Figure 4 not only generates a map $\Sigma$, but it also computes a dependency graph $\mathcal{G}$ on memory values. (Considering only the formalization of this section, the memory values of the graph are actually left-values only. However, when considering separately the ACSL predicates `\initialized` and `\valid` instead of `defined`, this is not true anymore.) This graph is necessary for ensuring, first, the soundness of the rule system with respect to mutual dependency on left-values in $\Sigma$, and, consequently, for the correct ordering of left-value initializations when generating C code (*cf.* Section 4).

Generally speaking, each time a rule that needs inference is used in a state constraint derivation for some left-value $L$ (*e.g.* DEREFERENCE, RANGE-1, CMP-1, *etc.*), edges from $L$ to every other left-value involved in some premise are added to the dependency graph $\mathcal{G}$. Such derivation fails as soon as this latter operation makes the graph $\mathcal{G}$ cyclic.

*Example* When applying the inference system on our example in Figure 1, the final map associates the integer `length` to $[16, 16672] \oplus \{\texttt{RTC}(\texttt{length}\%16 \equiv 0)\}$ and the array `input` to $[0, \texttt{length} - 1] \oplus \emptyset$, along with the dependency graph in Figure 5.

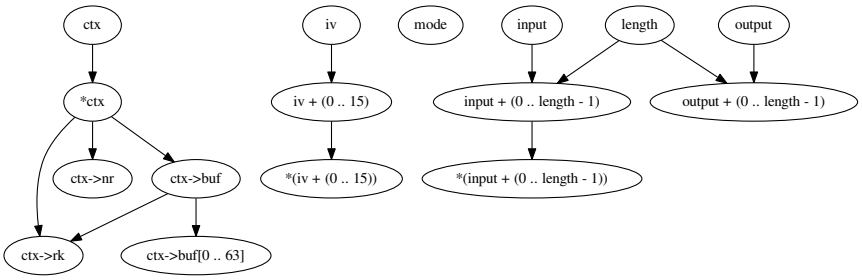


**Fig. 5:** Dependency graph for the `aes_crypt_cbc` preconditions generated by CfP.

The system of inference rule in Figure 4 is sound: given a conjunctive clause $P$, the simplification procedure on $P$ always terminates, either with $\Sigma$ or it fails. In the former case, for each left-value $L$ in $P$, state constraints in $\Sigma$ satisfy respective literals in $P$ (that we denote as $\Sigma \models P$).

**Theorem 1.** *For all conjunctive clause $P$, either $\emptyset \vdash P \Rightarrow \Sigma$ and $\Sigma \models P$, or it fails.*

## 4 Generating C Code from State Constraints

This section presents the general scheme for implementing preconditions, through state constraints, in a C language enriched with one primitive function for handling ranges. In practice, such primitive is meant to be analyzer-specific so as to characterize state constraints as precisely as possible. As an example, we report on the case of our tool CfP. However, for the sake of conciseness, we do neither detail nor formalize the code generation scheme. We nevertheless believe that the provided explanation should be enough to both understand and implement such a system in a similar setting.

*Generating Code from a Conjunctive Clause* Consider a conjunctive clause $\mathcal{C}$ and the pair $(\Sigma, \mathcal{G})$, respectively given by the map of state constraints and the dependency graph of $\mathcal{C}$, inferred by the system of rules in Figure 4. We shall show the general case of disjunctive normal forms $\bigvee_{i=1}^{n} \mathcal{C}_i$ later on.

To generate semantically correct C code, we topologically iterate over the left-values of $\mathcal{G}$ so as to follow the dependency ordering. For every visited left-value $L$, we consider its associated state constraint $C = R \oplus X$ in $\Sigma$. Then, the symbolic range $R$ is handled by generating statements that initialize $L$. For most constructs, these statements are actually a single assignment, although a loop over an assignment may be sometimes needed (*e.g.* when initializing a range of array cells). In particular, initializations of left-values $L$ to symbolic ranges $[T_1, T_2]$ are implemented by means of the primitive function $\texttt{make\_range}(\kappa, T_1, T_2)$, where $\kappa$ is integer or pointer type. In practice, this function must be provided by the analyzer for which the context is generated, so that, when executed symbolically, the analyzer's abstract state will associate abstract

values $[T_1, T_2]$ to respective left-values $L$. Finally, conditionals are generated to initialize left-values with symbolic expressions involving `min` and `max`.

Once $L$ has been initialized, the rest of the code is guarded by conditionals generated from runtime checks in $X$. To resume, the generation scheme for $L$ is the following:

```
/* initialization of L from R through assignments */
if (/* runtime checks from X */) {
  /* code for initializing the next left-values */ ...; } }
```

After the initialization of the last left-value, the function under consideration (in our running example, the function `aes_crypt_cbc`) is called with the required arguments.

*Handling Disjunctions* We rewrite preconditions into disjunctive normal form $\bigvee_{i=1}^{n} \mathcal{C}_i$ as a preliminary step. Then we process each disjunct $\mathcal{C}_i$ independently by applying the inference system in Figure 4 and the code generation scheme previously described.

We now describe the code generation scheme of such a precondition $\bigvee_{i=1}^{n} \mathcal{C}_i$ given the code fragments for each and every of its disjunct $\mathcal{C}_i$. If $n = 1$, then the code fragment of $\mathcal{C}_1$ is directly generated. Otherwise, an additional variable `cfp_disjunction` is generated and initialized to the interval $[1, n]$. Then, a `switch` construct (or a conditional if $n = 2$) is generated, where each case contains the fragment $B_i$ respective to $\mathcal{C}_i$. To resume, the context is generated as a function including the following code pattern:

```
cfp_disjunction = make_range(ι, 1, n);
switch (cfp_disjunction) {
  case 1: { B_1; break; }
  case 2: { B_2; break; }
  ...
  case n: { B_n; break; }
}
```

*Primitives in CfP* Our tool CfP follows the generation scheme just described. It implements `make_range` in terms of the Frama-C built-ins `Frama_C_`$\tau$`_interval`, with $\tau$ a C integral type, and `Frama_C_make_unknown` to handle symbolic ranges for integers and pointers, respectively. These built-ins are properly supported by the two abstract interpretation-based value analysis tools EVA [3] and TIS-Analyzer [8].

## 5 Implementation and Evaluation

We have implemented our context generation mechanism as a Frama-C plug-in, called CfP for *Context from Preconditions*, written in approximately 3500 lines of OCaml. (Although Frama-C is open source, CfP is not, due to current contractual obligations.) CfP has been successfully used by the company TrustInSoft for its verification kit [21] of the mbed-TLS library, an open source implementation of the SSL/TLS protocol.

We now evaluate our approach, and in particular CfP, in terms of some quite natural properties, that is, *usefulness*, *efficiency*, and *quality* of the generated contexts.

This work provides a first formal answer to a practical and recurring problem when analyzing single functions. Indeed, the ACSL subset considered is expressive enough for most real-world C programs. Most importantly, CfP enables any tool to support a compelling fragment of ACSL at the minor expense of implementing two Frama-C built-ins, particularly so if compared to the implementation of a native support (if ever possible). Finally, CfP has proved useful in an operational industrial setting in revealing some mistakes in contexts previously written by hand by expert verification engineers.

Although we cannot disclose precise data about these latter, CfP revealed, most notably, overlooked cases in disjunctions and led to fix incomplete specifications.

CfP is able to efficiently handle rather complex ACSL preconditions: the generation of real-world contexts (*e.g.* the one of Figure 2) is usually instantaneous. Although the disjunctive normal form can be exponentially larger than the original precondition formula, such transformation is used in practice [17,12] and leads to better code in terms of readability and tractability by the verification tools. This approach is further justified by the fact that, in practice, just a small number of disjuncts are typically used in manually-written ACSL specifications.

Our approach allows to generate contexts which are reasonably readable and follows code patterns that experts of the Frama-C framework use to manually write. In particular, when handling disjunctions, CfP factorizes the generated code for a particular left-value as soon as the rule system infers the very same solution in each conjunctive clause. For instance, in our running example, only the initialization of the variable `mode` depends on the disjunction `mode == 0 || mode == 1`. Hence all the other left-values are initialized before considering `cfp_disjunction` (*cf.* Figure 2).

We conclude by briefly discussing some current limitations. Our ACSL fragment considers quantifier free predicate formulæ, and no coercion constructs are allowed. Support for casts among integer left-values should be easy to add, whereas treating memory addresses as integers is notoriously difficult. We leave these for future work.

## 6 Related Work

Similarly to our approach, program synthesis [12,20,16] automatically provides program fragments from formal specifications. However, the two approaches have different purposes. Once executed either symbolically or concretely, a synthesized program provides *one* computational state that satisfies the specification, while a context must characterize *all* such states. In particular, not only every state must satisfy the specification but, conversely, this set of states must contain every such possible one.

In software testing, contexts are useful for concentrating the testing effort on particular inputs. Most test input generation tools, like CUTE [19] and PathCrawler [5,9], allow to express contexts as functions which, however, the user must manually write. Some others, like Pex [1], directly compile formal preconditions for runtime checking.

The tool STADY [15] shares some elements of our approach. It instruments C functions with additional code for ensuring pre- and postconditions compliance, allowing monitoring and test generation. However, the tool performs a simple ACSL-to-C translation, it does neither take into account dependencies among C left-values, nor it inferences their domain of definition.

## 7 Conclusion

This paper has presented a novel technique to automatically generate an analysis context from a formal precondition of a C function. The core of the system has been formalized, while we provide enough details about code generation to allow similar systems to be implemented. Future work includes the formalization of code generation as well as statements and proofs of the fundamental properties of the system as a whole. A running example from the real world has also illustrated our presentation. The whole system is

implemented in the Frama-C plug-in CfP. It generates code as close as possible to human-written code. It is used in an operational industrial setting and already revealed some mistakes in contexts previously written by hand by expert verification engineers.

## Acknowledgments

Part of the research work leading to these results has received funding for the S3P project from French DGE and BPIFrance. The authors thank TrustInSoft for the support and, in particular, Pascal Cuoq, Benjamin Monate and Anne Pacalet for providing the initial specification, test cases and insightful comments. Thanks to the anonymous reviewers for many useful suggestions and advice.

## References

1. M. Barnett, M. Fähndrich, P. de Halleux, F. Logozzo, and N. Tillmann. Exploiting the synergy between automated-test-generation and programming-by-contract. In *ICSE'09*.
2. P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language*. `http://frama-c.com/acsl.html`.
3. S. Blazy, D. Bühler, and B. Yakobowski. Structuring Abstract Interpreters through State and Value Abstractions. In *VMCAI'17*.
4. W. Blume and R. Eigenmann. Symbolic Range Propagation. In *IPPS'95*.
5. B. Botella, M. Delahaye, S. H. T. Ha, N. Kosmatov, P. Mouy, M. Roger, and N. Williams. Automating Structural Testing of C Programs: Experience with PathCrawler. In *AST'09*.
6. G. Canet, P. Cuoq, and B. Monate. A Value Analysis for C Programs. In *SCAM'09*.
7. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL'77*.
8. P. Cuoq and R. Rieu-Helft. Result graphs for an abstract interpretation-based static analyzer. In *JFLA'17*.
9. M. Delahaye and N. Kosmatov. A Late Treatment of C Precondition in Dynamic Symbolic Execution. In *CSTVA'13*.
10. ISO. The ANSI C standard (C99). Technical Report WG14 N1124, ISO/IEC, 1999. `http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1124.pdf`.
11. F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A Software Analysis Perspective. *Formal Aspects of Computing*, 2015.
12. V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete Functional Synthesis. In *PLDI'10*.
13. F. Logozzo and M. Fähndrich. Pentagons: A Weakly Relational Abstract Domain for the Efficient Validation of Array Accesses. In *SAC'08*.
14. H. Nazaré, I. Maffra, W. Santos, L. Barbosa, L. Gonnord, and F. M. Quintão Pereira. Validation of Memory Accesses Through Symbolic Analyses. *SIGPLAN Not.*, 49(10), 2014.
15. G. Petiot, B. Botella, J. Julliand, N. Kosmatov, and J. Signoles. Instrumentation of Annotated C Programs for Test Generation. In *SCAM'14*.
16. N. Polikarpova, I. Kuraj, and A. Solar-Lezama. Program Synthesis from Polymorphic Refinement Types. In *PLDI'16*.
17. W. Pugh. A Practical Algorithm for Exact Array Dependence Analysis. *Comm. ACM*, 1992.
18. R. Rugina and M. Rinard. Symbolic Bounds Analysis of Pointers, Array Indices, and Accessed Memory Regions. In *PLDI'00*.
19. K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *FSE'13*.
20. A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Sketching Stencils. In *PLDI'07*.
21. TrustInSoft. PolarSSL 1.1.8 verification kit, v1.0. Technical report. `http://trust-in-soft.com/polarSSL_demo.pdf`.