Institute of Informatics
Ludwig–Maximilians–Universität München

Bachelor's Thesis

Converting between ACSL Annotations and Witness Invariants

Sven Umbricht

Supervisor:	Prof. Dr. Dirk Beyer
Mentor:	Martin Spießl
Date of Submission:	December 14, 2020

Statement of Originality

I hereby confirm that I have written the accompanying thesis myself, without contributions from any sources other than those cited in the text and acknowledgements. This applies also to all graphics, tables, and images included in the thesis.

Munich, December 14, 2020 ....

Sven Umbricht

Abstract

Proving the correctness of a given program with regard to a certain specification is hard. To make this task easier one can additionally give invariants that may aid the verification. Several formats exist to provide invariants for this purpose, like GraphML-based correctness witnesses or ACSL, but herein already lies the problem: A tool that relies on having invariants provided in a specific format cannot profit from invariants that are structured differently. It would therefore be helpful to be able to translate invariants from one format into the other. The goal of this thesis is to translate invariants from correctness witnesses into ACSL annotations and vice versa. We describe a possible way to perform these translations and implement a proof of concept in CPAchecker. Our evaluation shows that we can indeed generate valid ACSL-annotated programs from correctness witnesses produced by different verifiers and that we are able to again create valid witnesses for these annotated programs.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Within the last two decades it has become common practice that tools for automatic software verification do not only output a pass/fail verdict for a given verification task, but supplement this result with some kind of proof [5]. This way the results of an untrusted verifier can easily be confirmed or discarded by a second, trusted one that only has to validate that the proof is indeed correct, which is usually a much simpler task than verifying the original program again [13]. This has been taken even further: Instead of producing a proof for the whole verification run, the first verifier can also just supply the validator with some hints to reach the same verification result on its own. For example, instead of giving a proof that a certain property holds for a given loop it is enough to provide a suitable loop invariant and let the validator construct the proof on its own. This strategy is often still cheaper than having the validator verify the program again from scratch, because confirming the correctness of a given invariant can be much faster than finding it was [4]. Many exchange formats already exist to convey such hints, but without a unified approach it can happen that verification results, even if they come with some kind of proof skeleton, have to be validated from scratch again because the validator does not support the same format. That is quite unfortunate, as often at least some parts of such a proof skeleton could be represented in the format used by the other tool.

In this thesis we will give an example of a translation between two formats for such verification helpers, more specifically the specification language used by the Frama-C [12] framework, ACSL, and the exchange format for witnesses [4, 5] that is used in the International Competition on Software Verification (SV-COMP) [2]. After an introduction of the covered features of these exchange formats we are going to look at how the proof guiding hints of one can be translated and represented in the other. To show the practical applicability of these translation concepts, an implementation integrated into CPAchecker [9], a tool that until now supported witnesses but not ACSL, will be presented and evaluated. An overview of the known issues as well as possible improvements and extensions will close this thesis paper.

Chapter 2

Preliminaries

Before we delve into the main topic of this thesis we will lay out the foundations necessary to follow the upcoming explanations of translation concepts. In this chapter we are going to give an overview of ACSL, the witness format and CPAchecker. We will also make some assumptions and introduce a few conventions we are going to use throughout the rest of this thesis.

2.1 ACSL

ACSL[1] is a contract-based specification language for the C99 version of the programming language C. It is inspired by the specification language of Caduceus which is itself inspired by the probably more commonly known JML for Java programs and is being developed by CEA LIST and INRIA Saclay – Île-de-France as the specification language for the Frama-C framework[2]. For this thesis we will use ACSL version 1.14 as described in the ACSL Manual of the Frama-C 20.0 (Calcium) implementation [1].

ACSL contracts (or ACSL annotations, these terms will be used interchangeably in this thesis) can be used to verify program properties and are usually placed directly in the program file in the form of C comments starting with an @ sign, i.e., `//@ contract` or `/*@ contract */`. For the meaning of an annotation it does not matter whether it is embedded in a single line or multiline comment. The *scope* of an annotation is the code for which the property described by the annotation should hold. This is usually the statement or block of statements directly following the annotation. This means that the position of an annotation is important, because properties do not generally hold at every program location. Therefore, position information would still have to be retained even if the annotations were to be stored separately. In this thesis we will always consider ACSL contracts to be stored as comments inside the program, but this does not actually matter for any of the presented translation concepts. The *pre-state* of

[1]The acronym ACSL is shorthand for "**A**NSI/ISO **C** **S**pecification **L**anguage".
[2]https://frama-c.com/index.html

a contract is the program state right before the first statement in the scope of the contract is executed. Similarly, the *post-state* of a contract is the program state directly after the last statement in the scope of the contract was executed. Whenever we refer to *loops* we will be talking only about explicit loops, i.e., for-, while-, and do-while-loops.

In the following we are going to introduce the building blocks of many of ACSL's different types of annotations as well as some of its most common kinds of contracts.

2.1.1 Logic Expressions

The basic building blocks of ACSL annotations are so called *logic expressions* that roughly correspond to C expressions but also include some additional constructs that can not be expressed in pure C. In the grammar for logic expressions (Fig. 2.1) we distinguish between *terms* and *predicates*, which behave similar to terms and predicates from classical first-order predicate logic. Note that this is a simplified version of the grammar given in the ACSL Manual that will produce only a subset of all valid logic expressions. There are a few additional constructs that we will not cover anywhere else in this thesis, so we omit them here as well.

An important thing to note is that logic expressions only have total functions, meaning that terms like $x/0$ are well-defined and can be used in annotations. Each logic expression also has a type. The type of a logic expression can be either a C type, a mathematical type, or a self-defined type. Mathematical types are *integer* and *real* for non-overflowing integers and real numbers respectively, as well as *boolean* for the boolean values `\true` and `\false`.

Additionally to the logic expressions defined by the grammar, there are a number of built-in logic expressions of varying arity like `\at(x, label)`, `\old(x)` or `\empty`. The semantics of these built-ins are usually not expressible through pure C expressions. For example, `\old(x)` refers to the value of expression `x` in the pre-state of the current contract. We will not occupy ourselves with built-in logic expressions much and will only focus on two built-ins: `\at` and `\result`. The `\at(x, label)` construct has the value that the logic expression `x` had the last time when the program location where the Label `label` is placed was reached. `label` can be either a C label placed in the program or a predefined label. Predefined labels can be used in ACSL annotations even without being declared in the program and are the following:

- Pre: Refers to the pre-state of the current function.
- Post: Refers to the post-state of the current function.
- Old: Refers to the pre-state of the current contract.
- Here: Refers to the location of the current contract. In function contracts (cf. Sect. 2.1.3) it is equivalent to Post when used in ensures-clauses (cf. Sect. 2.1.2) and equivalent to Pre when used in any of the other clauses that we cover.

```
⟨log-op⟩ ::= && | || | ==> | <==> | ^^

⟨bit-op⟩ ::= & | | | --> | <--> | ^

⟨rel-op⟩ ::= == | != | <= | >= | > | <

⟨arith-op⟩ ::= + | - | * | / | % | << | >>

⟨unary-op⟩ ::= + | - | ~ | * | &

⟨term⟩ ::= ⟨literal⟩
  |  ⟨id⟩
  |  ⟨unary-op⟩ ⟨term⟩
  |  ⟨term⟩ ⟨bit-op⟩ ⟨term⟩
  |  ⟨term⟩ ⟨arith-op⟩ ⟨term⟩
  |  ⟨term⟩ [ ⟨term⟩ ]
  |  ( ⟨type-expr⟩ ) ⟨term⟩
  |  ( ⟨term⟩ )
  |  sizeof ( ⟨term⟩ )

⟨pred⟩ ::= \true
  |  \false
  |  ⟨term⟩ (⟨rel-op⟩ ⟨term⟩)+
  |  ( ⟨pred⟩ )
  |  ⟨pred⟩ ⟨log-op⟩ ⟨pred⟩
  |  ! ⟨pred⟩
  |  ⟨pred⟩ ? ⟨pred⟩ : ⟨pred⟩
```

Figure 2.1: Grammar of logic expressions. <id> stands for an identifier, e.g., a variable name, <literal> for an integer- or a string-literal, and <type-expr> for an expression that evaluates to a C type. If multiple relational operators are "chained", e.g., $x\ op_1\ y\ op_2\ z$ then the resulting predicate is equivalent to $x\ op_1\ y$ `&&` $y\ op_2\ z$. The operators all have to "face in the same direction", i.e. the chain must be monotonous, so the set of used operators has to be a subset of either {`<, <=, ==`}, {`>, >=, ==`}, or {`!=`}.

- LoopEntry: Only usable in loops. Refers to the state right before entering the loop for the first time. When used in a nested loop, it refers to the start of the innermost loop.
- LoopCurrent: Only usable in loops. Refers to the beginning of the current loop iteration. When used in a nested loop, it refers to the current iteration of the innermost loop.
- Init: Refers to the program state right before the main function is called.

```
/*@ requires x >= 0;
    ensures \result * \result <= x < (\result + 1) * (\result + 1);
*/
int sqrt(int x) {...}
```

Figure 2.2: A contract containing a requires- and an ensures clause

`\old(x)` is actually just a shortcut for `\at(x, Old)`, meaning that it will also be covered implicitly. The other built-in we will include, `\result`, is much simpler: It refers to the current function's return value.

2.1.2 Clauses

Logic expressions hold truth values (in the case of predicates) or describe elements of a C program (in the case of terms), like pointers, literals or functions. However, they alone do not describe a property that could be confirmed by another verifier. To define properties ACSL uses various kinds of clauses. Clauses start with a keyword identifying the kind of clause and are followed by a logic expression, either a term or a predicate depending on the kind of clause. One contract can have any positive number of clauses, even multiple ones of the same kind. Multiple clauses of the same type in one annotation act like a single clause of that type where the contained logic expressions are joined via conjunction. Due to that we will assume in the following that every contract contains at most one clause of any given type. Some examples of clause types:

Ensures clauses Ensures clauses begin with the `ensures` keyword and declare that the contained logic expression (a predicate) holds once the scope of the annotation is left. Variables in ensures clauses refer to the variable value in the post-state of the contract they are used in.

Requires clauses Requires clauses begin with the `requires` keyword. They demand that the annotated code is only entered in a state where the logic expression contained in the clause (a predicate) evaluates to `\true`. Variables in requires clauses refer to their pre-state values, i.e., their values at the start of the annotated code. Using `\old(x)` in requires clauses would therefore be useless and is in fact not allowed at all.

An example usage of requires- and ensures clauses is given in Fig. 2.2 in the form of a specification for a function that computes the integer square root of its input.

Behaviors Sometimes clauses only hold under certain circumstances. While this could be expressed via a ternary condition, ACSL allows defining so called *behaviors* for more general applicability and better readability. Behaviors each have a name and contain a number of clauses that only have to hold under a certain condition. The condition that has to be met is given via an assumes

```
/*@ ensures x >= 0;
    behavior positive:
        assumes x > 0;
        ensures \result == x;
    behavior negative:
        assumes x < 0;
        ensures \result == -x;
    behavior zero:
        assumes x == 0;
        ensures \result == 0;
*/
int abs(int x) {...}
```

Figure 2.3: A contract with multiple behaviors

clause. As one might expect, assumes clauses begin with the `assumes` keyword and the contained logic expression has to be a predicate. Only if this predicate evaluates to true when the annotated code is entered have the other clauses of the behavior to hold. If no assumes clause is present in a behavior it acts as if an `assumes \true;` were present, meaning that the contained clauses always have to hold. Just like for all other kinds of clauses, multiple assumes clauses are equivalent to a single assumes clause containing the conjunction of their logic expressions. In order to give different conditions multiple behaviors can be specified in one contract. An example for a contract with multiple behaviors can be seen in Fig. 2.3.

Behaviors can also be reused by other contracts: Certain kinds of annotations may address behaviors defined in surrounding annotations by their names and add their own clauses that only have to hold if the condition from the assumes clause of the addressed behavior held when the surrounding annotation was entered.

Completeness clauses Related to behaviors are completeness clauses. These can be used to ensure the correctness of a set of behaviors. Completeness clauses begin either with `complete behaviors` or with `disjoint behaviors` and are optionally followed by a comma-separated list of behavior names. If `complete behaviors` is used at least one behavior's condition has to be fulfilled whenever the annotated code is entered. If `disjoint behaviors` is used the conjunction of the assumes clauses of any two listed behaviors has to be unsatisfiable, meaning that at most one behavior can be active at once. A completeness clause without any behaviors specified acts as if all behaviors of the current annotation had been given. (This is true for both types of completeness clause.) For example, the annotation from Fig. 2.3 could be extended with both a `complete behaviors` and a `disjoint behaviors` clause, because the assumptions $x > 0$, $x < 0$, and $x == 0$ cover all possible cases and are mutually exclusive.

Additional clauses Other clause types include for example `assigns` clauses that hold terms describing the variables modified in the annotated code, `terminates` clauses that contain a condition under which the annotated code terminates and `decreases` clauses that hold terms containing expressions whose value is decreased by the annotated code. However, in this thesis we will limit ourselves to the clause types mentioned above, i.e., ensures-, requires-, assumes-, and completeness clauses, as well as behaviors.

2.1.3 Annotations

ACSL does not only support different kinds of clauses to describe program properties but also provides different types of annotations, each with their own semantics and scoping rules. Technically most kinds of annotations could also be expressed via one or more ACSL assertions, but the diversity of annotation types makes it possible to write annotations that are more concise and better understandable for a human reader. Having different annotation types can also make it easier to handle them, e.g., because the scope of an annotation is already clear from the kind of annotation used. We will limit ourselves to four common types of annotations which we are going to present here:

Function Contracts The most important type of ACSL annotation is the function contract [14]. Function contracts are used to specify the properties of a function. They are placed right before the function or function declaration and just as is the case with multiple clauses of the same type in one annotation, multiple function contracts for the same function are treated like a single contract containing the union of clauses and behaviors. Function contracts may contain all of the clause types mentioned above, though assumes clauses are only allowed to appear inside of behaviors.

Statement Contracts Statement contracts are similar to function contracts in that they also use clauses and behaviors to specify program properties. However, statement contracts do not describe properties of a function, but, as one might guess, those of a statement. The grammar for statement contracts is therefore very similar to the grammar for function contracts but the contract has to be placed directly before the statement in question. Statement contracts may use requires-, ensures-, assigns-, and completeness clauses as well as behaviors, but no clauses that only make sense on the function level, like terminates clauses.

A feature of statement contracts that is not present in function contracts is the possibility to extend behaviors of enclosing annotations. An enclosing annotation is an annotation for a block that the statement contract is contained in. The behavior of such an enclosing annotation can then be extended by specifying that the statement contract only has to hold if the assumes condition of the behavior was met when the block annotated by the enclosing annotation was entered. An example for this can be seen in Fig. 2.4.

Loop Annotations A bit more specialized than statement contracts are loop annotations. These specify properties of a loop and the contained clauses are

```
/*@ behavior even:
        assumes x % 2 == 0;
*/
int foo(int x) {
    int y;
    /*@ for even:
            ensures y * 2 == x;
    */
    y = x / 2;
    ...
}
```

Figure 2.4: A statement contract referring to an enclosing behavior

```
int x = 0;
int y = 10;
//@ loop invariant x + y == 10;
while (++x < 10) {
    y--;
}
```

Figure 2.5: A valid loop invariant that does not hold directly after the loop

slightly different from those used in other contracts. A clause unique to loop annotations is the loop invariant. ACSL loop invariants contain a predicate I that acts as an invariant for the annotated loop. I has to hold right before the loop is entered, that is, right before the loop condition is checked (or, in the case of a do-while loop, before the first statement in the loop is executed) and has to hold again after each loop iteration that ended normally, that is, not via a break or return statement or a goto jumping out of the loop. An ACSL loop invariant does not necessarily have to hold after the loop is left though because checking the condition might have side-effects, as demonstrated in Fig. 2.5.

Aside from loop invariants, loop annotations may also contain loop-assigns clauses which are similar to normal assigns clauses, and loop variants. Loop annotations may also refer to surrounding behaviors, like statement contracts do, and contain additional clauses that only have to hold if the condition of the referenced behavior was met.

Assertions ACSL assertions are usually short annotations and are allowed before any kind of statement or at the end of a block before the closing bracket. They contain no clauses at all, only a predicate that is expected to evaluate to `\true` at the program location where the assertion is placed. Assertions may start with either the `assert` or the `check` keyword, the only difference being that check-assertions are not interrupting program execution even if the predicate is found to be false. Just like statement contracts or loop annotations, assertions may refer to behaviors of surrounding annotations and in this case only have

```
...
while (x > 0) {
    //@ assert x > 0;
    y++;
    x--;
    //@ assert x >= 0;
}
//@ assert x <= 0;
...
```

Figure 2.6: A loop with some valid assertions

```
int main(void) {
    int x = 0;
    int y = x;
    while (x < 10) {
        y--;
        x++;
    }
    if (x + y != 0) {
        ERROR: return 1;
    }
    return 0;
}
```

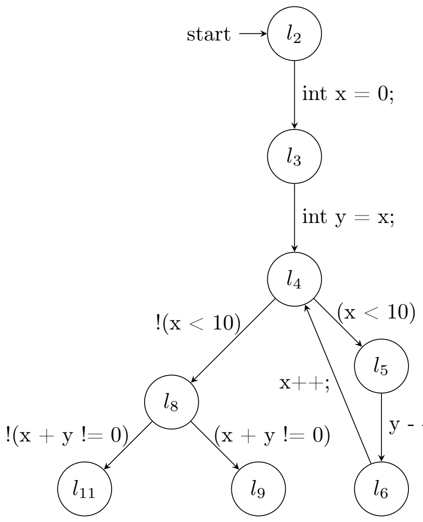

Figure 2.7: A program and its control flow automaton

to hold if one of the referenced behaviors' assumes clause predicate was true. Examples for valid ACSL assertions can be seen in the code snippet from Fig. 2.6.

2.2 Control Flow Automaton

A program can be represented by its *control flow automaton* (CFA). Formally, we regard a CFA as a tuple (L, pc_0, G) where L is the set of program locations, $pc_0 \in L$ is the initial location, and $G \subseteq L \times Ops \times L$ is the set of control-flow edges with Ops being the set of operations that can be performed when taking one of these edges [8]. We can represent the CFA of a program as a directed graph whose nodes represent the current value of the program counter (e.g., the line number of the next operation to be performed) and whose edges represent the program statements. Figure 2.7 exemplarily shows a program and the corresponding CFA.

2.3 Exchangeable Witness Format

The exchangeable witness format is an exchange format for witness automata [4, 5] that observe or guide the program path exploration of a validator. Witness automata are protocol automata: A protocol automaton A for a CFA $C = (L, pc_0, G)$ is a five-tuple $(Q, \Sigma, \delta, q_0, F)$ where Q is the (final) set of automaton states, $q_0 \in Q$ is the initial state, the alphabet $\Sigma \subseteq 2^G \times \Phi$ is a set of pairs, each containing a set of control-flow edges and a condition $\phi \in \Phi$, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, and F is the set of accepting states.

There are two types of witnesses, violation witnesses and correctness witnesses. Violation witnesses guide the program path exploration by adding *state-space guards* to the automaton transitions. A transition can only be taken if the condition imposed by the state-space guard is fulfilled. This way the number of program paths that can be explored is reduced and the validator can find a property violation more easily. Correctness witnesses do not restrict the number of explorable program paths and merely observe the exploration effort. They may provide invariants that hold at certain points during the exploration and might reduce the amount of work the validator has to do [4]. In this thesis we will focus solely on correctness witnesses, since ACSL is also meant to help with proofing correctness.

Since correctness witnesses only observe the program path exploration without interfering, correctness witness automata are observer automata. An observer automaton $O = (Q, \Sigma, \delta, q_0, F)$ for a CFA $C = (L, pc_0, G)$ is a protocol automaton such that for all $q \in Q$ and for all $g \in G$:

$$\bigvee \{\phi | \exists q' \in Q : \exists \sigma \in \Sigma : \exists D \subseteq G : (q, \sigma, q') \in \delta \land \sigma = (D, \phi) \land g \in D\} = true$$

This means that the conditions on the transitions may not restrict the program path exploration, or simply that for every edge taken in the CFA there is a possible transition in the observer automaton. Violation witnesses for instance are not observer automata since they can make such restrictions via state-space guards.

The exchange format for witnesses[3] is based on GraphML [10]. A witness file contains a graph representation of a witness automaton, where nodes represent states and edges represent transitions. Both nodes and edges of a graph described in a witness file may be enriched with additional information via GraphML-Attributes: This is done by adding a *data* element as a child of the node/edge that should be annotated with additional information. *data* elements have a *key* attribute that identifies the kind of data contained. In the case of correctness witnesses, valid keys and the corresponding contents of the *data* element are:

- for children of nodes:
  - *entry*: contains true or false; if true: the node represents the initial state of the witness automaton

[3] The exchange format is being maintained at `https://github.com/sosy-lab/sv-witnesses`

    - *invariant*: contains an invariant for the state represented by the node
    - *invariant.scope*: contains the program scope of the variables in invariants at the node

- for children of edges:
    - *control*: contains condition-true or condition-false; the transition represented by the edge only matches a CFA edge that corresponds to a branching in the source code where the branching condition evaluates to true (in the case of condition-true) or to false (in the case of condition-false)
    - *startline*: contains a line number that should match the line where the statement of a CFA edge starts
    - *endline*: contains a line number that should match the line where the statement of a CFA edge ends
    - *startoffset*: contains a character offset that should match the offset of the first character of the statement of a CFA edge
    - *endoffset*: contains a character offset that should match the offset of the last character of the statement of a CFA edge
    - *enterLoopHead*: contains true or false; if true: the transition represented by the edge only matches a CFA edge that enters a loop head
    - *enterFunction*: contains the name of the function that is entered by the transition represented by the edge
    - *returnFromFunction*: contains the name of the function that is left by the transition represented by the edge

For our use-case we are mainly interested in *data* elements containing invariants. Invariants in a witness must be valid C expressions that evaluate to C type int[4] and contain no function calls. A sibling *data* element with key *invariant.scope* can be used to qualify variables in the invariant in order to prevent mapping of the invariant variables to the wrong variables from the program. There are currently some limitations to this mechanism so we will not rely on it in the following chapters; however, being able to give more specific scopes in the witness would be one way to support the representation of some ACSL constructs that can currently not be expressed in a witness so this mechanism could prove useful in a future version of the witness format. Data elements describing program locations (keys startline, endline, startoffset, endoffset) can be useful in determining the positions of ACSL annotations derived from witness invariants in the program.

[4]Witness invariants are used as predicates just as one might expect. However, since logical operators and comparison operators in C return zero for false and one for true, an invariant evaluates to an integer value instead of a boolean one. (Similarly, C implicitly converts zero to false and any non-zero integer to true.)

2.4 Configurable Program Analysis

A configurable program analysis (CPA) defines the abstract domain of a program analysis [6]. We can formally describe a CPA as a four-tuple $(D, \rightsquigarrow, \mathit{merge}, \mathit{stop})$, where D is the abstract domain, $\rightsquigarrow$ is the transfer relation, *merge* is the merge operator, and *stop* is the termination check. These components are described in the following.

Abstract Domain The abstract domain $D = (C, \mathcal{E}, [\![\cdot]\!])$ consists of the set of concrete states C, the semi-lattice of abstract states $\mathcal{E}$, and the concretization function $[\![\cdot]\!]\colon E \to 2^C$ that maps each abstract state to the set of concrete states that it represents. The semi-lattice $\mathcal{E} = (E, \sqsubseteq, \sqcup, \top)$ consists of the set of elements E, the reflexive and transitive less-than relation $\sqsubseteq \subseteq E \times E$, the join operator $\sqcup\colon E \times E \to E$ which is a total function that is defined for two abstract states e and e' as the smallest $e'' \in E$ with $e \sqsubseteq e''$ and $e' \sqsubseteq e''$, and the top element of the lattice $\top \in E$ which is the most abstract state with regard to $\sqsubseteq$, i.e., $e \sqsubseteq \top$ holds for every $e \in E$.

Transfer Relation The transfer relation $\rightsquigarrow \subseteq E \times G \times E$ defines for an abstract state $e \in E$ and a CFA edge $g \in G$ the set $\{e' \mid (e, g, e') \in \rightsquigarrow\}$ of abstract successor states of e for g.

Merge Operator The merge operator $\mathit{merge}\colon E \times E \to E$ returns an abstract state that combines the information of the two input states. The returned state is at least as abstract as the second input state, i.e., $e' \sqsubseteq \mathit{merge}(e, e')$ for every e, $e' \in E$.

Termination Check The termination check $\mathit{stop}\colon E \times 2^E \to \mathbb{B}$ determines whether the abstract state given as first parameter is covered by the set of states given as second parameter.

An example for a CPA is the Composite CPA. The Composite CPA can combine a number of other CPAs such that they can exchange information with each other, which makes it feasible to have several simple CPAs that each only track one specific aspect of the whole analysis. Formally a Composite CPA that combines CPAs $C_1, \ldots, C_n$ where $C_i = (D_{C_i}, \rightsquigarrow_{C_i}, \mathit{merge}_{C_i}, \mathit{stop}_{C_i})$ can be represented as follows:

- The abstract domain is the cartesian product of the abstract domains of its components.
- The transfer relation yields as abstract successors the cartesian product of the abstract successors of the component CPAs. This means that the Composite CPA contains exactly one abstract successor iff all of its components have exactly one abstract successor, and that there is no successor iff any of the components has no successor.

- The merge operator merges states componentwise, i.e., it returns the cartesian product of the merged components.
- The termination check calls the termination checks of the component CPAs and returns true iff all of the components returned true.

Of course, being configurable this is only one way to define the Composite CPA.

CPAs can be used by the CPA Algorithm (c.f. Alg. 1) to determine the set of reachable abstract program states. The set *reached* contains said reachable states while the set *waitlist* contains those states that have been discovered as being reachable but have not yet been processed. Both sets are initialized with the initial abstract state given as input to the algorithm. Then, while *waitlist* is not empty, an element from *waitlist* is picked and its abstract successor states are computed. For each of these successors a *merge* operation is then performed with every state that has been reached so far. Should the merge yield a new element the old one is replaced because the new state is more abstract than the old one (recall the definition of *merge*) and therefore all concrete states that are described by the old state are also described by the new one, i.e., the old one is covered. Every successor state is also added to the sets *waitlist* and *reached*, as long as the termination check does not request otherwise. The time taken for the analysis and the abstraction level of the results depend on each of the used CPA's components.

Algorithm 1 CPA Algorithm
(*adapted from: Combining Model Checking and Data-Flow Analysis* [7])

Input: a CPA (D, $\rightsquigarrow$, *merge*, *stop*), an initial abstract state $\mathrm{e}_0 \in E$ where E is the set of elements of the lattice of D
Output: the set of reachable abstract states
Variables: a set reached $\subseteq E$, a set waitlist $\subseteq E$

```
1:  waitlist := $\{\mathrm{e}_0\}$
2:  reached := $\{\mathrm{e}_0\}$
3:  while waitlist $\neq \emptyset$ do
4:      choose e from waitlist
5:      waitlist := waitlist $\setminus \{\mathrm{e}\}$
6:      for all $\mathrm{e}'$ with $\mathrm{e} \rightsquigarrow \mathrm{e}'$ do
7:          for all $\mathrm{e}'' \in$ reached do
8:              $\mathrm{e}_{new} := merge(\mathrm{e}', \mathrm{e}'')$
9:              if $\mathrm{e}_{new} \neq \mathrm{e}''$ then
10:                 waitlist := (waitlist $\cup \{\mathrm{e}_{new}\}) \setminus \{\mathrm{e}''\}$
11:                 reached := (reached $\cup \{\mathrm{e}_{new}\}) \setminus \{\mathrm{e}''\}$
12:         if $\neg$ stop($\mathrm{e}'$, reached) then
13:             waitlist := waitlist $\cup \{\mathrm{e}'\}$
14:             reached := reached $\cup \{\mathrm{e}'\}$
15: return reached
```

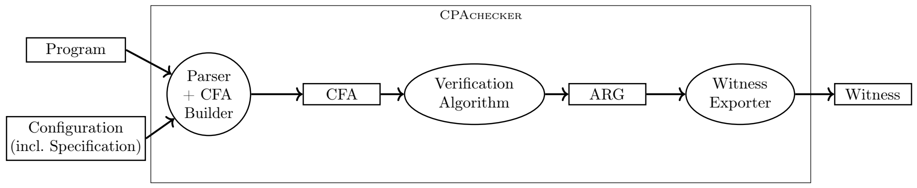

Figure 2.8: Overview of Witness Generation in CPAchecker

2.5 CPAchecker

CPAchecker [9] is a framework for configurable program verification of C programs that is able to create witnesses as a way to validate verification results. The process of producing a witness for a given program with CPAchecker is illustrated in Fig. 2.8. It can be split into three steps: First the program is parsed and a CFA for the program is built. Then the actual verification commences, working with the CFA rather than the original program. For this step CPAchecker can make use of different CPAs, depending on the configuration provided by the user alongside the program, to compute an abstract reachability graph (ARG). The ARG contains the reachable abstract states of the program as well as their relations. Finally a witness is built using the information from the ARG and a graph representation of the witness is exported as a GraphML file.

Chapter 3

Converting between ACSL and Witness Invariants

In order to make CPAchecker compatible with tools understanding or producing ACSL annotations two mechanism have to be added to CPAchecker: Extracting invariants usable by CPAchecker (and other tools that support the witness format) from ACSL annotations and creating ACSL annotations from the invariants of a witness. This chapter gives an overview over which kinds of ACSL constructs can be used or generated and describes a possible translation between ACSL annotations and witness invariants.

3.1 Invariants from ACSL Annotations

ACSL is a powerful language with a large amount of elements. It is beyond the scope of this thesis to provide support for translating every kind of construct ACSL has to offer. In this work we will only describe methods to translate the four most common types of ACSL annotations which were already introduced above (cf. Sect. 2.1.3). In fact, for most of these annotation types we will merely provide means to represent a subset of all valid annotations of that type and will therefore work with simpler grammars than those found in the ACSL Manual [1] just like we did when introducing logic expressions (cf. Sect. 2.1.1). These restrictions are often due to the much lower expressiveness of the witness format compared to ACSL and are detailed in the corresponding subchapter for each annotation type. Possible ways to increase the number of translatable annotations are discussed in Chapter 6.

Before looking into the annotation-specific translations in detail we are going to present some observations that are useful for the translation of several different annotation types. First off, note that most ACSL predicates can already be considered syntactically valid witness invariants. As mentioned before (cf. Sect. 2.1.1), logic expressions in ACSL are often valid C expressions, and since predicates additionally evaluate to either `\true` or `\false` they meet all necessary requirements to be usable as invariants in a witness. Since witness invariants may contain conjunctions and disjunctions we can also utilize logical equivalences to transform some ACSL predicates that are not C expressions into valid invariants. In the grammar for logic expressions that we will work with (cf. Fig. 2.1) we already excluded all features that can not be expressed via pure C expressions.

Another thing to note is the fact that, as mentioned before (cf. Sect. 2.1.3), most properties described in annotations can also be expressed by a number of ACSL assertions. Of course, these assertions can in general not just be placed where the original annotation was, but have to be moved to appropriate locations. For example, the ensures clause `ensures E;` of a function- or statement contract can be replaced by an assertion `/*@ assert E;*/` at every program location directly before the scope of the contract is left. How exactly to split up the different kinds of annotations will be demonstrated in the upcoming subchapters.

After these observations it is only a small step to the idea of how to translate annotations to invariants. We first decompose an annotation into one or multiple simple ACSL assertions of the form `assert` P_i`;` and if necessary transform the predicates P_i into valid invariants. Each invariant then holds at the program location where the corresponding assertion would be placed, too. In the following, we will talk about the predicates P_i rather than the assertions and will call them *predicate representations* of the original annotation.

Before showing how to construct predicate representations for annotations, we are going to introduce predicate representations for behaviors. Behaviors can also be represented by predicates and it is possible to use their predicate representations in the predicate representations of the annotations they are used in. Recall that a behavior may contain assumes-, requires- and ensures-clauses. Since the variables from the assumes- and requires-clauses refer to variable values in the pre-state of the annotation that the behavior is part of, and the variables from the ensures-clauses refer to values from the post-state of said annotation, we create two predicates to represent a behavior.

Let X be the set of valid ACSL contracts and behaviors and Y the set of ACSL predicates that can be transformed into valid witness invariants. We define two functions $pre\colon X \mapsto Y$ and $post\colon X \mapsto Y$ that map a contract or behavior to its *pre-state (predicate) representation* and its *post-state (predicate) representation*, respectively. For a behavior, the pre-state representation shall represent the requires-clause and the post-state representation shall represent the ensures-clause. Because the assumes-clause contains the condition under which the behavior is active and neither requires- nor ensures-clauses have to hold if the behavior is not active in the first place, the assumes-clause has to be considered in both the pre-state representation and the post-state representation. Therefore, for a behavior B of the form

```
/*@ ...
behavior B:
    assumes A;
    requires R;
    ensures E;
... */
```

the equivalence

$$pre(B) \iff R \vee \neg A$$

has to hold. For the post-state representation however, the formula

$$post(B) \iff E \vee \neg A$$

does not hold in general because A does no longer have to hold at the position where E would hold, since they do not refer to variable values at the same location. Instead, the ACSL built-in predicate `\at` has to be used. Since `\at` is a predicate that can not be represented in a C expression, a suitable translation has to be used. In the following we will use *at* for an abstract way of representing the translated `\at`. For example, the equivalence

$$post(B) \iff E \vee \neg at(A, Old)$$

for the post-state representation of a behavior does hold, because it refers to the value of A in the correct program state. It is not necessary to talk about where the assertions holding the predicate representations have to be placed in the program; this depends entirely on the annotation in which the behavior is located, as behaviors can never appear outside of an annotation in the first place.

There are several ways to implement *at*. One possibility is to use *data* elements with the key "assumption": For every unique $at(x, label)$ that appears in a predicate representation, the witness-automaton transitions for the corresponding CFA edge are duplicated and one of the duplicate transitions gets the additional child *<data key="assumption">x</data>* while the other gets the additional child *<data key="assumption">!x</data>*. The reachable part of the witness automaton after those transitions would also have to be duplicated and in the duplicate following the first transition the expression $at(x, label)$ could be replaced with *true* while in the other $at(x, label)$ could be replaced with false. This duplication is necessary so that the witness does not restrict the explorable state space.

A different approach would be to assign additional constants in the program: A copy of the original program could be verified instead, after the values of the used $at(x, label)$ constructs were stored in fresh constants. These constants would be instantiated at an appropriate location where the value of $at(x, label)$ is accessible even in the C program, e.g., at the beginning of a function for the built-in label *Pre*. Since their value never changes, every $at(x, label)$ could then be replaced with the appropriate constant.

The first approach has the advantage of leaving the program untouched but the witness can get very big since it grows exponentially with the number of *at* constructs used in predicate representations. The key "assumption" is also currently not allowed in correctness witnesses, but if used as described the properties of a correctness witness would still be kept intact. The second approach has no drawbacks regarding the witness, except that it does not really create a witness for the original program. The modified program would also have to be written out and used when validating the witness.

3.1.1 Invariants from Function Contracts

Function contracts from which invariants should be created may contain only requires-, ensures- and completeness clauses, as well as behaviors. Other kinds of clauses, like assigns- or terminates clauses will not be discussed in detail, because predicates representing them are too unwieldy. For example, a clause `assigns a;` could be represented by $b_1 == at(b_1, Old) \wedge \ldots \wedge b_n == at(b_n, Old)$ where $\{b_i | 1 <= i <= n\}$ is the set of all variables existing in both the pre- and the post-state of the annotation wherein the clause occurs, except for `a`.

Just like for behaviors, in order to produce an invariant from a function contract we create a pre-state representation and a post-state representation. The pre-state representation of a function contract shall contain the predicate from the requires-clause and the pre-state representations of its behaviors, while the post-state representation shall contain the predicate from the ensures-clause and the post-state representation of its behaviors. Completeness clauses are not included in either predicate because they would not contribute anything when used in invariants. After all, they do not contain information about the program but about the annotation itself. As a result, for a function contract F of the form

```
/*@ requires R;
    ensures E;
    behavior B_1: ...
    ...
    behavior B_n: ...
*/
```

we have the following equivalences:

$$\begin{aligned} pre(F) &\iff R \wedge pre(B_1) \wedge \cdots \wedge pre(B_n) \\ post(F) &\iff E \wedge post(B_1) \wedge \ldots \wedge post(B_n) \end{aligned}$$

Since the properties in the pre-state representation have to hold whenever the function is entered, we can just consider $pre(F)$ as an invariant at the start of the function. Similarly, as the properties from the post-state representation have to hold every time the function terminates we can use $post(F)$ as an invariant that holds at every function exit. An example for the correct placement of invariants can be seen in Fig. 3.1.

3.1.2 Invariants from Statement Contracts

Just like function contracts, statement contracts from which invariants should be extracted may contain requires-, ensures- and completeness clauses, as well

```
/*@ requires x < y;
    ensures z == x + y;
*/
int sum(int x, int y) {
    while (x < y) {
        x++;
        y--;
    }
    int z = x * 2;
    if (x == y) {
        return z;
    }
    z--;
    return z;
}
```

(i)

```
int sum(int x, int y) {
    // invariant: x < y
    while (x < y) {
        x++;
        y--;
    }
    int z = x * 2;
    if (x == y) {
        // invariant: z == x + y
        return z;
    }
    z--;
    // invariant: z == x + y
    return z;
}
```

(ii)

Figure 3.1: A function with (i) an ACSL function contract (ii) a schematic representation of where which invariants extracted from the contract hold.

as behaviors. With this reduced number of allowed ACSL constructs the only syntactic difference left between function contracts and statement contracts is that statement contracts are allowed to have to hold only under certain behaviors of enclosing function- or statement contracts.

Since supported function- and statement contracts are so similar syntactically, the processes of generating invariants from them can also be very similar to each other. Again we create a pre-state and a post-state representation that represent the same clauses as the pre- and post-state representation of a function contract, only that this time the relevant enclosing behaviors are also included: Since a statement contract S

```
/*@ for C_1, ... , C_m:
        requires R;
        ensures E;
        behavior B_1: ...
        ...
        behavior B_n: ...
*/
```

only has to hold if the assumption from at least one referenced behavior C_i held in the pre-state of the corresponding enclosing annotation, the equivalences

$$pre(S) \iff (\neg at(A_1, old_1) \wedge \ldots \wedge \neg at(A_m, old_m)) \vee (R \wedge pre(B_1) \wedge \ldots \wedge pre(B_n))$$

$$post(S) \iff (\neg at(A_1, old_1) \wedge \ldots \wedge \neg at(A_m, old_m)) \vee (E \wedge pre(B_1) \wedge \ldots \wedge pre(B_n))$$

shall hold for the pre-/post-state representation of a statement contract, where A_i is the predicate from the assumes-clause of the referenced behavior C_i and old_i is a label placed at the pre-state of the annotation wherein C_i is placed. There is no pre-defined logic label to refer to the position of old_i, so it is unlikely that this construct can ever be used in practice because it would require that a C label is placed at the appropriate location in the program. For statement contracts

```c
int i = 0;
int j = 10;
//@ loop invariant i + j = 10;
while (j > 0) {
    i++;
    j--;
}
```

(i) A program excerpt with an ACSL loop annotation...

```c
int i = 0;
int j = 10;
// invariant: i + j == 10
while (j > 0) {
    i++;
    j--;
    // invariant: i + j == 10
}
```

(ii) ...and with a schematic representation of the appropriate locations for the extracted invariant

Figure 3.2: Extracting witness invariants from ACSL loop invariants

that do not refer to enclosing behaviors, the pre- and post-state representations shall be the same as for function contracts. In either case the pre-state predicate representation has to hold directly before, the post-state representation directly after the statement, so they can be used as invariants for the corresponding location in the program.

3.1.3 Invariants from Loop Annotations

Loop annotations for the purpose of invariant generation may only contain loop invariants, though these are allowed to be either general or specific to certain enclosing behaviors. Since a loop invariant already contains a predicate we can just use that as an invariant for the program, meaning that it is not necessary to differentiate between a pre- and a post-state representation. In the case of a loop invariant specific to certain enclosing behaviors the same construct as for statement contracts can be used, i.e., the form

$$(\neg at(A_1, old_1) \wedge \ldots \wedge \neg at(A_m, old_m)) \vee I$$

where I is the predicate from the loop invariant, the A_i are the conditions of the relevant enclosing behaviors and the old_i are labels placed at the pre-state of the contract wherein said behaviors appear. Again, fittingly placed labels are usually not present in the program, so this construct would seldom appear in practice and usually only simple loop invariants of the form `loop invariant I;` can be represented in witnesses. In any case, the generated invariant holds right before the loop and after each loop iteration (for a do-while loop this means after every condition check), but not necessarily after the loop is left as this is not guaranteed by the ACSL invariant. An example for the correct placement of the derived invariants is shown in Fig. 3.2.

3.1.4 Invariants from Assertions

For the translation of assertions there are no further restrictions: a valid witness invariant can be extracted from every valid ACSL assertion. In fact, the predicate contained in the assertion may already be used as an invariant itself. If the assertion is specific to certain enclosing behaviors the same strategy as for statement contracts and loop annotations can be applied. In either case, the invariant generated from the assertion holds at the same program location where the assertion was placed, too.

One could argue that assertions introduced with the `check` keyword are incapable of providing any invariant at all, since they do not interrupt program execution even if they turn out to be invalid and therefore can be seen as not necessarily having to hold. On the other hand, assertions are expected to hold even if they are introduced with the `check` keyword so treating `assert`-assertions and `check`-assertions equally is also a valid option. In the implementation detailed in Chapter 4 we are going to choose the latter option so as to not potentially lose useful information.

3.2 ACSL Annotations from Witness Invariants

There are two things we need in order to create an ACSL-annotated program from the original one and a set of witness invariants: A way to translate the invariants into ACSL annotations and the means to determine the correct positions of those annotations.

Since invariants taken from a witness are valid C expressions and therefore valid ACSL predicates the conversion from invariant to ACSL annotation is much simpler than the other direction because it is possible to just wrap every invariant in an ACSL assertion. Multiple assertions created this way can then be merged into a single statement- or function contract or a loop annotation. Depending on the positions where an invariant holds, it could also be used in the ensures-clause of a function contract or as an ACSL loop invariant directly. However, such approaches only serve to reduce the number of annotations and should not yield more information.

The only difficulty for this direction is extracting the correct location for the ACSL annotation from the witness. There are several ways to achieve this: A simple approach would be to use the location information stored in the witness file, e.g., the startoffset and endoffset values stored for the witness transitions. Since these refer to character offsets of program statements, an ACSL assertion derived from a witness invariant can be placed at all program locations between the endoffset values of entering and startoffset values of leaving transitions of the state where the invariant is placed. For example, consider a witness containing the following:

```
...
<node id="N2">
    <data key="invariant">( x == y )</data>
    <data key="invariant.scope">main</data>
</node>
<edge source="N1" target="N2">
    <data key="startoffset">45</data>
    <data key="endoffset">62</data>
</edge>
<node id="N3"/>
<edge source="N2" target="N3">
    <data key="startoffset">74</data>
    <data key="endoffset">98</data>
</edge>
...
```

Then the assertion `/*@ assert (x == y); */` could be placed anywhere between character offsets 62 and 74 in the original program, assuming there are no other entering or leaving transitions present for node N2.

Another possibility to get an appropriate location for the ACSL assertion is to perform a reachability analysis on the program's CFA while simultaneously following the appropriate transitions in the witness automaton. The assertion could then be placed at all program locations that were visited while the witness automaton remained in the state with the invariant.

Both ways have their own advantages and disadvantages: The first approach is easier to implement and does not even need access to the program because the information can just be extracted from the witness, however there is no guarantee that there are any *data* elements with keys startoffset or endoffset present in the first place. The second approach does not rely on optional information being present in the witness file and can be used with every valid witness. It does, however, require the program to be available and is slower due to the additional parsing and reachability analysis of the program. This approach can also produce unnecessarily many ACSL annotations if the witness automaton remains in the state containing the invariant for several consecutive CFA edges. Both approaches have the drawback that it is not clear whether the invariant always holds at the determined location or only on certain paths, since the invariants specified in a witness do not have to be location invariants. We expect that in practice most invariants are location invariants though and will not try to find a way to express more general invariants in this thesis. A different approach where that would be possible is outlined in Chapter 6.

Once it has been decided how exactly the conversion to ACSL annotations and the determination of their correct location should be performed, the $WitnessToACSL$ Algorithm given in Alg. 2 can be applied. Essentially, the algorithm iterates over the list of program statements and copies each statement into the new program P', inserting the converted invariants where deemed appropriate. Any invariants that were not added to P' once the entire program was copied this way are then appended at the end of P' and the annotated program is returned. Apart from some standard operations on lists, sets, and natural numbers, we only need two additional functions: `location` and `toACSL`, possible implementations of which were discussed above.

There is one nontrivial condition imposed on `toACSL` though: `toACSL` may not introduce dependencies between different annotations at the same location because the order in which they are appended to the program is non-deterministic. The only way that such a dependency could be introduced is by converting one invariant into a statement contract with a behavior and translating another invariant into a statement contract referencing that behavior though, which would be a very rare occurrence. In our implementation this cannot happen at all and we only mention this restriction here for completeness.

Algorithm 2 WitnessToACSL Algorithm

```
Input: a set $I$ of invariants, a program $P$ as a list of program statements
Output: the annotated program $P'$ as a list of program statements
Variables: a set converted $\subseteq I$, a counter n $\in \mathbb{N}$
 1: converted := $\emptyset$
 2: n := 0
 3: while n < $P$.length() do
 4:     J := $\{$i $\in I$ | n $\in$ location(i)$\}$
 5:     for all j $\in$ J do
 6:         $P'$.append(toACSL(j))
 7:     $P'$.append($P$.get(n))
 8:     converted := converted $\cup$ J
 9:     n := n + 1
10: for all k $\in I \setminus$ converted do
11:     $P'$.append(toACSL(k))
12: return $P'$
```

Chapter 4

Implementation

An implementation of the translations discussed in Chapter 3 has been integrated into CPAchecker. For the translation from ACSL-annotated program to witness we extend the parsing process of CPAchecker to also parse any ACSL annotations present in the program file and implement a CPA dedicated to providing invariants based on ACSL annotations. However, we do not provide a concrete translation of the `\at` construct. CPAchecker can represent annotations containing `\at` internally but does not export them as witness invariants. For the generation of ACSL-annotated programs from witnesses we provide an algorithm that takes invariants from a witness, turns them into ACSL annotations and inserts them into the program file at an appropriate location.

4.1 Utilizing ACSL Annotations in CPAchecker

In order to make use of ACSL annotations CPAchecker has to parse and turn them into a useable format first, in this case into an object of type `ExpressionTree`. In this implementation the annotations are not immediately turned into `ExpressionTree`s however, but stored as `ACSLAnnotation` objects. Should an `ExpressionTree` be required, e.g., for writing into a witness, the conversion will be made and the result provided by the `ACSLCPA`.

In the following subchapters we will take a closer look at the process of utilizing ACSL annotations in CPAchecker.

4.1.1 Parsing ACSL Annotations with CPAchecker

CPAchecker uses the Eclipse CDT for parsing C programs. The Eclipse CDT parser creates an abstract syntax tree (AST) of the program that is then traversed by an `ASTVisitor` that collects the information necessary to build the CFA of the program. We extend the traversal process to also inspect `IASTComment` objects, which correspond to the comments in the program file. If a comment has the format of an ACSL annotation (i.e., it starts with either /*@ or //@) its location is mapped to the CFA edges corresponding to the program locations directly in front of and behind the comment. Once the CFA of the program has been created, we use a CUP-generated parser together with a tokenizer generated by JFlex to parse the ACSL annotations in a second iteration over the program file, this time ignoring everything but ACSL annotations. In this step we make use of the previously generated mapping and the CFA in order to correctly interpret annotations. For example, the annotation `/*@ ensures x == 10; */` by itself could be understood as either a function- or a statement contract. With the additional information from the first round of parsing the correct annotation type can be determined.

Finally, we create a mapping from CFA edges to those ACSL annotations whose invariant representations would have to hold at the location directly after taking the edge, and store it alongside the CFA. The invariants are not being generated at this point but the locations can already be determined as has been described above (cf. Sect. 3.1).

4.1.2 Representing ACSL Annotations in CPAchecker

Once they are parsed, CPAchecker stores representations of ACSL annotations as objects of type `ACSLAnnotation`. The interface `ACSLAnnotation` provides only two methods: `getPredicateRepresentation()` and `getCompletenessPredicate()`. Both return an object of type `ACSLPredicate` which represents a predicate as defined in the grammar (Fig. 2.1) and can be translated into an `ExpressionTree` by calling `toExpressionTree()`. `getCompletenessPredicate()` is merely a utility method to make sure that the completeness clause of an annotation is met as otherwise the annotation is faulty and shouldn't be used for invariant generation. `getPredicateRepresentation()` returns the predicate representation of the annotation as described in Sect. 3.1. For function- and statement contracts the pre- or post-state representation is returned depending on the current program location. This is achieved by keeping a copy of the contract for every CFA edge entering or leaving its scope and storing whether the copy is mapped to an entering or a leaving edge. The obtained representation is then usable as an invariant once translated via `toExpressionTree()`.

4.1.3 The ACSL CPA in CPAchecker

To be able to not only use invariants obtained from ACSL annotations in witnesses but also in other parts of CPAchecker a CPA for providing these invariants has been added. Since the translation of an ACSL annotation to a valid invariant is already implemented elsewhere, the CPA acts merely as a distributor of those invariants: The `ACSLTransferRelation` when asked for successors for a CFA edge always provides a single `ACSLState` that stores those ACSL annotations whose invariant representations have to hold directly after the edge is taken. The class `ACSLState` itself implements the interface `ExpressionTreeReportingState` by giving an implementation of `getFormulaApproximation()` that returns the conjunction of the invariants produced from the stored ACSL annotations. This way any component of CPAchecker that can use invariants provided by an `ExpressionTreeReportingState` can profit from them.

4.2 Building ACSL Annotations in CPAchecker

The conversion from witness invariants to ACSL annotations is realized in the `WitnessToACSLAlgorithm` which implements Alg. 2. For this we make use of CPAchecker's already existing `WitnessInvariantsExtractor` that is capable of scanning a witness for invariants and returning them in the form of `ExpressionTreeLocationInvariant` objects that also store the CFA node where the invariant is placed. The `ExpressionTreeLocationInvariant`s are then mapped to the program locations where they have to hold. To do so, we consider all leaving edges of the CFA node stored in an `ExpressionTreeLocationInvariant` and extract from each the line number of the corresponding program statement. The invariants are then converted into ACSL assertions as described in Sect. 3.2 and the assertions placed one line above each statement.

This has the obvious disadvantage that the program file may only contain one statement per line or else the assertion may be placed too early, so programs that should get ACSL annotations this way might have to be reformatted beforehand. In fact, the reformatting would have to happen before generating the witness because otherwise the position information recorded in the witness might no longer match the actual positions in the program.

Chapter 5

Evaluation

In this chapter we will examine the correctness and usefulness of the translation concepts described above by evaluating if and how many of the produced ACSL-annotated programs can still be verified and whether the witnesses containing invariants derived from ACSL-annotations can still be validated. In particular, the hypotheses we hope to confirm are:

1. We can generate syntactically correct ACSL annotations from correctness witnesses.
2. Generated annotations can be validated by tools supporting ACSL.
3. ACSL annotations can be represented in CPAchecker and written out as invariants in witnesses.
4. Invariants in a witness that were created from ACSL annotations can be validated by tools that understand the witness format.

Should the translation from ACSL annotations to witness invariants be working sufficiently well, it would be possible to manually guide a verifier by writing ACSL annotations into the program and having them translated and written into the witness. Technically it is possible to write the invariants into the witness directly, but this is often not practicable since witness states do not necessarily correspond to program locations one-to-one and witnesses can get very big.

5.1 Experimental Setup

For our evaluation we use witnesses from the witness store of SV-COMP 2020[1] and the corresponding programs from the sv-benchmarks repository[2]. We only use correctness witnesses that contain at least one invariant, in total 10387 witnesses

[1]https://zenodo.org/record/3630188
[2]https://zenodo.org/record/3633334

and the programs they were created for (there might be multiple witnesses for a single program). The following evaluation steps are then performed, corresponding to the hypotheses listed above:

1. Run the WitnessToACSLAlgorithm to produce ACSL annotated programs.
2. Use Frama-C to try and prove generated ACSL annotations.
3. Create new witnesses from the annotated programs with a configuration that uses the ACSLCPA.
4. Validate the new witnesses with CPAchecker.

Since the WitnessToACSLAlgorithm only produces assertions we will not be able to directly evaluate the translations for other ACSL constructs this way. However, since other types of annotations can be expressed via assertions as well, this approach is still able to give a good impression of how well the translation between ACSL annotations and witnesses works. We are also not bothered by the lack of a concrete translation for `\at` because the WitnessToACSLAlgorithm does not introduce behaviors.

Each benchmark is executed on a machine running Ubuntu 20.04 and has access to 2 CPU cores with 3.40 GHz (Intel Xeon E3-1230 v5) and 15 GB of RAM. Each run also has a time limit of 900 s (15 min). For the evaluation we use CPAchecker on branch acsl-bachelor in revision 36000 and the Frama-C 20.0 (Calcium) release, as well as BenchExec[3] for benchmarking.

5.2 Results

ACSL Annotations from Witness Invariants For the first step of our evaluation we take the 10.387 correctness witnesses from SV-COMP 2020 together with their respective programs and pass them to the WitnessToACSLAlgorithm. Because the WitnessInvariantsExtractor that we use to obtain invariants from the witnesses actually performs a reachability analysis we use the `-skipRecursion` option in order to ignore recursive calls so that we can get annotations even for recursive programs which are not supported by CPAchecker per default. The results can be seen in Table 5.1. In total there are 612 witnesses for which the algorithm does not complete successfully due to timeouts, errors, or memory shortages. Upon closer inspection we can determine that none of the thrown exceptions are caused by the actual algorithm given in Alg. 2 but rather happen in other parts of CPAchecker. For example, all of the tasks with result EXCEPTION suffered from a StackOverflowError with most of them being caused by too deep recursion of a function *buildConditionTree* which has been known to exhibit this behavior for years[4]. Given the simplicity of the algorithm it is also likely that most of the TIMEOUT and OUT OF JAVA MEMORY results can be

[3]`https://github.com/sosy-lab/benchexec`
[4]`https://gitlab.com/sosy-lab/software/cpachecker/-/issues/504`

	count
done	9775
TIMEOUT	465
OUT OF JAVA MEMORY	112
ERROR	18
EXCEPTION	15
ASSERTION	2
sum	10387

Table 5.1: Results of WitnessToACSL Algorithm

attributed at least in part to the invariant-extraction process. For the remaining 9775 witnesses the WitnessToACSLAlgorithm terminates properly and we get output programs for 5387 witnesses. This number does not yet tell us how well our implementation of the algorithm works, since an output program is generated whenever at least one invariant was found by the WitnessInvariantsExtractor regardless of whether the algorithm actually adds any annotations. Instead we have to look at the number of generated programs that actually contain ACSL annotations which is 4685, so our implementation is often able to create annotations from the given invariants.

Validating the Generated ACSL Annotations Next we try to prove the generated annotations with Frama-C in order to make sure that they are actually valid and hold at the locations where they are placed. For this step we use Frama-C-SV[5], a wrapper for Frama-C that was developed so that Frama-C could be used as part of SV-COMP 2021. As input all of the 5387 programs produced in step one are used, even those that do not contain annotations. The latter will serve as a comparison set. The results are listed in Table 5.2. It has to be noted that Frama-C-SV is still rather unpolished and the results produced by it can therefore not be considered perfectly reliable. They can give a rough estimate of our translations validity and soundness though. We use revision 27853bad of Frama-C-SV with a slight modification: Usually Frama-C-SV would call Frama-C with the options `-val` and `-rte`. Since the latter instructs Frama-C to add additional annotations on its own which might interfere with our evaluation, we disable this option.

We see that most of the annotations can be proved, although there is also a large number of programs where Frama-C-SV is not able to come to a conclusion. Since this is also the case in the comparison set we will attribute this to the immaturity of Frama-C-SV. Just like CPAchecker, Frama-C does not support recursion by default which is why there are a few cases where Frama-C aborts because of it. There are also some cases where Frama-C is unable to complete the analysis because of a TIMEOUT or OUT OF MEMORY. This does never

[5] `https://gitlab.com/sosy-lab/software/Frama-C-SV`

	with annotations	without annotations	total
true	3188	340	3528
unknown	1445	340	1785
unknown(recursive)	17	22	39
TIMEOUT	22	0	22
OUT OF MEMORY	12	0	12
unknown(crash)	1	0	1
sum	4685	702	5387

Table 5.2: Results of ACSL Validation

happen in the comparison set and is likely due to the fact that witness invariants are sometimes very large and are not shrinked at all by our implementation, leading to equally large ACSL annotations. Of course it could also be that some of the affected programs simply contain annotations that Frama-C finds hard to proof. Finally there is one result with status "unknown(crash)" which seems to be an internal error of Frama-C and is most likely unrelated to the concrete annotations present in the corresponding program file.

All in all it seems that the annotations generated by our implementation of the WitnessToACSLAlgorithm are indeed syntactically correct and can often be proved by Frama-C, confirming the first two of our hypotheses.

Witnesses from Annotated Programs In the next step we try to generate new witnesses for the annotated programs created by the WitnessToACSLAlgorithm. Since we only used correctness witnesses for our original input we naturally assume that only correctness witnesses are created in this step. This time we do not use `-skipRecursion` because that could make the analysis unsound and we also set the option `cpa.acsl.ignoreTargetStates` to true. This ensures that all program locations that are affected by ACSL annotations are considered, meaning that every ACSL annotations gets translated into a witness invariant, but it also means that error states found by other CPAs are ignored. This should not be a problem for our evaluation, since the only other CPAs we use are the `LocationCPA`, `CallStackCPA`, and `FunctionPointerCPA` which are all used only to track certain information about the current state of the analysis.

The results of this step are displayed in Table 5.3. We see that most of the programs can still be verified, but also that there are quite a few programs for which we don't get a result due to various reasons that are all obviously caused by the ACSLCPA as they don't appear at all in the comparison set. The eight assertion errors marked with ASSERTION all happen while trying to match the ACSL annotations to the correct edges of the CFA. This tells us that the current approach of matching is not perfect yet, but it is already very good considering the small amount of failures. The 70 programs for which an EXCEPTION occurred all contain a . (period/dot) in one of their ACSL annotations. This is presumably because a floating point number is used which is not supported in the current implementation and does not indicate a problem. The "ERROR (recursion)" results are due to not using `-skipRecursion` as mentioned above and also appear in the comparison set, so they should be unrelated to the ACSLCPA. More worrying are the 296 results with status "ERROR (parsing failed)". Looking at the input programs we observe that this error always appears because of the same reason: All affected programs contain an annotation at the end of the program file, i.e., there is no other program statement following it, only closing brackets and whitespace. In this case our implementation is unable to determine the CFA edge corresponding to the program location directly behind the annotation, as described in Sect. 4.1.1, and ultimately fails when trying to determine the locations where the annotation holds from this information. It is unclear how such a case should be handled and we leave it as future work to find a way to rectify this. There is also a lot of programs for which a TIMEOUT occurs. This can be due to large annotations that were produced from large invariants, but since there are way more TIMEOUT results than during the validation of the generated ACSL annotations with Frama-C this is unlikely to be the only reason. A closer analysis of the programs for which a TIMEOUT occurs reveals that the creation of `ACSLAnnotation` objects is currently simply too inefficient, causing a TIMEOUT even for moderately sized annotations. This would of course have to be improved in the future but it is clear that this is only an implementation-specific flaw, since Frama-C only had a few results with status TIMEOUT, and is not inherent to the translation itself.

Finally we take note of the number of produced witnesses which is higher than the number of "true" results because CPAchecker apparently also produces witnesses for programs containing recursion even though they cannot be handled. Focusing only on those witnesses that actually contain invariants, we can see that there are unsurprisingly none at all in the comparison set because we did not use any CPA that is able to provide them apart from the ACSLCPA. Of the witnesses produced for annotated programs 1585 contain invariants. Since these have to be generated from ACSL annotations this confirms our third hypothesis. Considering that we started with witnesses that all contained at least one invariant though, this number seems rather mediocre. In 1844 of the 3483 cases were a witness was produced all of the annotations in the program were skipped because they contained an identifier that CPAchecker could not find a declaration for. This is often the case for annotations after a for-loop: An annotation placed directly after the loop might use the loop variable referring to its value after the last loop iteration, but in C the variable is already out of scope at this point. An example for this can be seen in Fig. 5.1. This particular problem can unfortunately not be resolved without modifying the program, since there is in general no program location where both the loop counter still exists and the invariant expressed by the annotation holds. CPAchecker can also be unable to find the declaration of an identifier if multiple candidates are found throughout the program. In this case no declaration is returned just like as if none was found. This could be alleviated by also including the scope in which the identifier occurs when trying to find its declaration; this way ambiguities especially in big files could be avoided. The remaining 54 witnesses might lack invariants because no

```
int i = 0;
for (int k = 10; k > 0; k--) {
    i++;
}
//@ assert k == 0 && i == 10;
```

Figure 5.1: The assertion accurately describes the state after the last loop iteration but is not valid because k is already out of scope.

	with annotations	without annotations	total
true	3392	663	4055
TIMEOUT	828	0	828
ERROR (parsing failed)	296	0	296
ERROR (recursion)	91	39	130
EXCEPTION	70	0	70
ASSERTION	8	0	8
sum	4685	702	5387
produced witnesses	3483	702	4185
witnesses with invariant	1585	0	1585

Table 5.3: Results of Witness Generation using ACSLCPA

annotation could be matched to a concrete program location, or because they were produced for programs containing recursion.

Validating the New Witnesses To ensure that the "true" results are actually correct we use CPAchecker to validate the witnesses produced in the second step. The result can be seen in Table 5.4 and is very straightforward: Almost all the witnesses can be validated, only for 23 inputs there is again an "ERROR (recursion)" result and one witness can not be validated because of a TIMEOUT. These results show that the ACSLCPA, or rather that invariants produced from ACSL annotations do not introduce any inconsistencies or break the witness format. We can therefore conclude that we are able to generate valid correctness witnesses from ACSL annotated programs using those annotations in witness invariants and furthermore, that our fourth hypothesis holds.

Manually annotating programs Since we have seen that our implementation of the presented translation works fairly well we will now try to combine the qualities of ACSL and the witness exchange format, namely being interactive and being exchangeable between different verifiers, respectively. To do so, we manually annotate a few sample programs with loop invariants that are known to be hard to find for automatic verifiers but can easily be found by a human. The programs are taken from *Software Verification with PDR: An Implementation of the State of the Art* [3]. Figure 5.2 exemplarily shows the relevant part of the

	with annotations	without annotations	total
true	3463	698	4161
ERROR (recursion)	19	4	23
TIMEOUT	1	0	1
sum	3483	702	4185

Table 5.4: Results of Witness Validation

```c
int main(void) {
    unsigned int w = __VERIFIER_nondet_uint();
    unsigned int x = w;
    unsigned int y = w + 1;
    unsigned int z = x + 1;
    //@ loop invariant y == z;
    while (__VERIFIER_nondet_uint()) {
        y++;
        z++;
    }
    __VERIFIER_assert(y == z);
    return 0;
}
```

Figure 5.2: The main-function of program eq2.c with an ACSL loop annotation. __VERIFIER_nondet_uint returns a random unsigned integer;__VERIFIER_assert simulates an assertion.

annotated version of program `eq2.c` from that paper. We then create witnesses for these annotated programs using the ACSLCPA and try to validate them with CPAchecker. The produced witnesses all contain an invariant that corresponds to the loop invariant specified via ACSL, and all of the witnesses can be validated in less than ten seconds. To make sure that these problems are indeed not trivial we also try to validate empty witnesses for these programs. As can be seen in Table 5.5 our previous claim that invariants are hard to come up with for verifiers holds true, since none of the witnesses can be validated without the invariants from the manually added ACSL annotations.

This means that our translation from ACSL annotations to invariants can even help verifying programs by enabling people to specify invariants manually. As stated in the beginning of this chapter, manually adding invariants to witnesses is possible in theory but tedious and often effectively impossible in practice. This problem can now be avoided by specifying invariants in the form of ACSL annotations in the program and afterwards translating these annotations into witness invariants.

program	with invariants	without invariants
bin-suffix-5.c	6.44s	T
const.c	6.87s	T
eq1.c	7.81s	T
eq2.c	7.41s	T
even.c	6.91s	T
mod4.c	7.11s	T
odd.c	7.28s	T

Table 5.5: Time taken to validate witnesses with and without invariants from ACSL annotations. T indicates a TIMEOUT, meaning that the program could not be validated within 900s.

Chapter 6

Future Work

An obvious extension would be to increase the number of supported ACSL features. Though we only worked with a subset of all available ACSL features so that we would be able to provide a possible translation for as many as possible, that does not mean that none of the excluded features can be translated in a meaningful way. In particular, for none of the excluded features did we give a formal proof that it can not be translated, so a formal evaluation of which ACSL futures can and cannot be translated would be an interesting area for future research. The evaluation of concrete translations of `\at` and other ACSL built-ins could also lead to a more seamless translation between ACSL annotations and witness invariants.

The current implementation of the `WitnessToACSLAlgorithm` could be improved to be applicable for C programs in general, without the need to ensure that each line contains at most one statement.

Our evaluation also revealed a few areas where the current implementation of the translation from ACSL annotation to witness invariant could be improved. This includes being able to handle annotations that are placed after the last statement in a program file, which is non-trivial at least for CPAchecker because of the way that branching is handled in the CFA. Another thing to work on is the efficiency of creating the internal representations of ACSL annotations. This issue especially should be taken care of, because it is the reason why a big number of tasks suffered from a TIMEOUT and because this can be improved very much as demonstrated by Frama-C. We also discovered that often annotations are simply skipped because CPAchecker cannot find the declaration of a variable it contains and therefore treats it as invalid. It was already discussed that this could probably be remedied in some cases by also taking note of the scope wherein the variable occurs, but that there are also cases where it is simply impossible to find a location for an annotation where all of the contained variables are in the current C scope.

The translation concepts covered in this thesis are merely possibilities. There certainly are other possible ways to translate ACSL annotations into witness invariants and vice versa. For example, one approach worth exploring is to use *ghost code* to simulate the transitions of a witness automaton. Ghost code is a feature of ACSL we did not cover in this thesis. Basically, it allows to add additional statements to a program in the form of ACSL annotations starting with the `ghost` keyword, i.e., a comment like `/*@ ghost ... */`. These ghost statements may not interfere with the semantics of the original program but are allowed to read the values of both ghost variables, i.e., variables declared in ghost code, and variables that are part of the original code. In order to simulate a witness automaton with ghost code it would be possible to track the current automaton state in a ghost variable by updating it after every statement that causes a transition to be taken in the automaton. Witness invariants could then be translated to ACSL annotations and the latter be placed at those locations where the ghost variable has the value that represents the automaton state where the invariant holds. An advantage of using ghost code to simulate the witness automaton is that even invariants that only hold on certain program paths can be translated by including the value of the ghost variable tracking the automaton state in the generated annotation.

The main reason we barely covered any ACSL built-ins at all is simply that the current witness format cannot express them adequately. This limitation is already known and a new version of the format has been suggested that might even support ACSL annotations natively. Should this new format end up being used in the future, the translation concepts presented in this thesis could be utilized to convert old witnesses into the new format, thus allowing validators to support only the newer version. To a lesser extent witnesses in the new format could also be backported into the old one should the need arise.

Chapter 7

Conclusion

We presented and explored a way to translate witness invariants to ACSL annotations and the other way around. We gave descriptions of how such translations could work in theory and discussed possible ways of implementing them. Some of those possibilities were then implemented in CPAchecker and evaluated using a large set of witnesses produced by several state-of-the-art verifiers. The results of our study show the practical applicability of the presented translation concepts but also open up new areas of research.

The WitnessToACSL Algorithm that we introduced to generate ACSL annotations from witness invariants has proven to work well even in our simple proof-of-concept implementation. The algorithm itself depends on three subroutines: one for translating the invariants to ACSL annotations, one for determining the location where the annotations should be placed, and (implicitly) one for supplying the invariants. All of these can be exchanged independently from another, meaning that there are several angles to improving this translation even further. The translation of ACSL annotations into witness invariants has produced promising results as well, but also showed some weaknesses. Our implementation is currently not efficient enough to handle annotations that contain more than a few hundred characters. It was already clear that not all ACSL annotations can be translated into valid witness invariants, because ACSL offers some constructs that can not be expressed in pure C. Because of this we restricted ourselves to a subset of ACSL that can mostly be expressed via witness invariants. We also discovered that some witness invariants do not correspond to a specific location in the program source meaning that ACSL annotations, even though they can be created, can not be placed anywhere in the program.

We have seen that our translation approach works well enough that users can manually specify invariants in the form of ACSL annotations to help with the verification. We were also able to prove a significant number of the ACSL annotations obtained from witness invariants. All in all we conclude that while our implementation can still be optimized, our approach is already able to allow the conversion between witness invariants and ACSL annotations in a wide variety of cases and has the potential to be improved even further.

Bibliography

[1] P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. ACSL Version 1.14 Implementation in 20.0 (Calcium).

[2] D. Beyer. Advances in Automatic Software Verification: SV-COMP 2020. In *Proceedings of the 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2020, Dublin, Ireland, April 25-30), part 2*, LNCS 12079, pages 347–367. Springer, 2020.

[3] D. Beyer and M. Dangl. Software Verification with PDR: An Implementation of the State of the Art. In *Proceedings of the 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2020, Dublin, Ireland, April 25-30), part 1*, LNCS 12078, pages 3–21. Springer, 2020.

[4] D. Beyer, M. Dangl, D. Dietsch, and M. Heizmann. Correctness Witnesses: Exchanging Verification Results Between Verifiers. In T. Zimmermann, J. Cleland-Huang, and Z. Su, editors, *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016, Seattle, WA, USA, November 13-18)*, pages 326–337. ACM, 2016.

[5] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer. Witness Validation and Stepwise Testification across Software Verifiers. In E. D. Nitto, M. Harman, and P. Heymans, editors, *Proceedings of the 2015 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on Foundations of Software Engineering (ESEC/FSE 2015, Bergamo, Italy, August 31 - September 4)*, pages 721–733. ACM, New York, 2015.

[6] D. Beyer, M. Dangl, and P. Wendler. A Unifying View on SMT-Based Software Verification. *Journal of Automated Reasoning*, 60(3):299–335, 2018.

[7] D. Beyer, S. Gulwani, and D. Schmidt. Combining Model Checking and Data-Flow Analysis. In E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors, *Handbook on Model Checking*, pages 493–540. Springer, 2018.

[8] D. Beyer, T. A. Henzinger, and G. Théoduloz. Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In W. Damm and H. Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 504–518. Springer, 2007.

[9] D. Beyer and M. E. Keremoglu. CPAchecker: A Tool for Configurable Software Verification. In G. Gopalakrishnan and S. Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011, Snowbird, UT, July 14-20)*, LNCS 6806, pages 184–190. Springer-Verlag, Heidelberg, 2011.

[10] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. S. Marshall. GraphML Progress Report. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Graph Drawing, 9th International Symposium, GD 2001 Vienna, Austria, September 23-26, 2001, Revised Papers*, volume 2265 of *Lecture Notes in Computer Science*, pages 501–512. Springer, 2001.

[11] J. Burghardt, J. Gerlach, T. Lapawczyk, et al. ACSL by Example, 2016. `https://github.com/fraunhoferfokus/acsl-by-example/blob/master/ACSL-by-Example.pdf`.

[12] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C - A Software Analysis Perspective. In G. Eleftherakis, M. Hinchey, and M. Holcombe, editors, *Software Engineering and Formal Methods - 10th International Conference, SEFM 2012, Thessaloniki, Greece, October 1-5, 2012. Proceedings*, volume 7504 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2012.

[13] S. Glesner. Program Checking with Certificates: Separating Correctness-Critical Code. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods, International Symposium of Formal Methods Europe, Pisa, Italy, September 8-14, 2003, Proceedings*, volume 2805 of *Lecture Notes in Computer Science*, pages 758–777. Springer, 2003.

[14] V. Prevosto. ACSL Mini-Tutorial. `https://frama-c.com/download/acsl-tutorial.pdf`.