Bachelor’s Thesis

Converting between ACSL Annotations and Witness Invariants

Sven Umbricht
Example program with loop invariant $b == a$
**Motivation (1)**

Example program with loop invariant \( b == a \)

```c
int main() {
    int a = 1, b = 1;
    while (b < 1000) {
        a += a;
        b += b;
    }
    if (a != b) {
        ERROR: return 1;
    }
    return 0;
}
```

GraphML-based correctness witness containing the invariant

```xml
...<node id="N9">  
  <data key="invariant">( b == a )</data>
  <data key="invariant.scope">main</data>
</node>  
<edge source="N2" target="N9">  
  <data key="enterLoopHead">true</data>
  <data key="startline">2</data>
  <data key="endline">2</data>
  <data key="startoffset">28</data>
  <data key="endoffset">32</data>
</edge>...```
Advantages of Annotations:

- Easy to understand/modify for a human
- No need for additional files
- Might create compatibility with other tools

```c
int main() {
    int a = 1, b = 1;
    //@ loop invariant b == a;
    while(b < 1000) {
        a += a;
        b += b;
    }
    if (a != b) {
        ERROR: return 1;
    }
    return 0;
}
```
Overview

- Preliminaries
- ACSL ⇔ Witness
- Evaluation
- Summary
ACSL

- **ANSI/ISO C Specification Language**
- Used by the Frama-C framework
- Specification as special comments in the program: /*@ . . . */ or //@ . . .
- Several kinds of annotations, e.g.
  - Function Contracts
  - Loop Annotations
  - Assertions
ACSL - Logic Expressions

- Building blocks of ACSL annotations
- Roughly correspond to C Expressions
- Distinction between *Terms* and *Predicates*, e.g.
  - $x$ and $1+2+3$ are terms
  - $\text{true}$ and $x == 0$ are predicates
ACSL - Assertions

▶ Structure: //@ assert <predicate>;

▶ Contained predicate should evaluate to true where the assertion is located

▶ Example:

```c
int x = 1;
//@ assert x == 1;
int y = 5;
//@ assert x + y < 10;
...```

ACSL - Function Contracts

▶ Specify properties of functions

▶ Made of different kinds of clauses, e.g.
  ▶ requires clauses describe properties of the pre-state
  ▶ ensures clauses describe properties of the post-state

▶ Example:

```c
/*@ requires y <= x;
   ensures x >= 0; */

int natural_subtraction(int x, int y) {
  ...
}
```
Correctness Witnesses

- Observe the state space exploration of the verifier
- May provide invariants that hold at certain program locations
- Invariants used in the GraphML-based witness exchange format
  - must be valid C expressions
  - must evaluate to an int
  - may contain conjunction/disjunction
  - may not contain function calls
Witness Invariants ⇒ ACSL Annotations

- Witness invariants are valid ACSL predicates
  → Conversion is easy

- Example:
  \[ x == 0 \text{ becomes } \text{assert } x == 0; \]

- But: Where to put assertions?
  - Use location information from witness
  - Run observer analysis on the program with the witness as observer automaton
ACSL Annotations ⇒ Witness Invariants (1)

- Basic idea: Represent annotations by predicates
- ACSL predicates are often equivalent to C expressions
- ACSL assertion can simply be represented by contained predicate → Conversion is straightforward
- Example:
  ```
  assert x ⇒ y; can be converted to !x // y
  ```
How to represent the following?

```c
/*@ requires y <= x;
   ensures x >= 0; */

int natural_subtraction (int x, int y) {
    x = x - y;
    return x;
}
```
Split up contract into several assertions:

```c
//@ requires y <= x;
ensures x >= 0; */
int natural_subtraction
  (int x, int y) {
  x = x - y;
  return x;
}
```

Translate assertions like before:

```c
int natural_subtraction
  (int x, int y) {
  //@ assert y <= x;
  x = x - y;
  //@ assert x >= 0;
  return x;
}
```
Implementation

[Diagram showing the flow of concepts from Specification to WitnessToACSLAlgorithm to AnnotatedProgram to CPA Algorithm to Witness, with additional paths to LocationCPA and ACSL CPA]
Evaluation - Goals

1. Generate valid ACSL annotations from correctness witnesses

2. Parse ACSL annotations and create witnesses containing derived invariants

3. Validate generated ACSL annotations/witnesses
Evaluation - Results (1)

Generate valid ACSL annotations from correctness witnesses

- Good performance of the actual algorithm ✓
- Often no result because invariants are not found ❌
- Found invariants can usually be converted successfully ✓

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>input witnesses</td>
<td>10387</td>
</tr>
<tr>
<td>algorithm done</td>
<td>9775</td>
</tr>
<tr>
<td>generated programs</td>
<td>5387</td>
</tr>
<tr>
<td>with annotations</td>
<td>4685</td>
</tr>
</tbody>
</table>
Evaluation - Results (2)

Parse ACSL annotations and create witnesses containing derived invariants

- ACSL annotations can be parsed and are interpreted correctly
- Parsing annotations is apparently inefficient
- Conversion is often possible and performed correctly
- Many annotations are skipped because they are invalid

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>input programs</td>
<td>4685</td>
</tr>
<tr>
<td>produced witnesses</td>
<td>3392</td>
</tr>
<tr>
<td>with invariant</td>
<td>1585</td>
</tr>
</tbody>
</table>
Evaluation - Results (2)

Parse ACSL annotations and create witnesses containing derived invariants

- ACSL annotations can be parsed and are interpreted correctly ✓
- Parsing annotations is apparently inefficient X
- Conversion is often possible and performed correctly ✓
- Many annotations are skipped because they are invalid X

<table>
<thead>
<tr>
<th>input programs</th>
<th>4685</th>
</tr>
</thead>
<tbody>
<tr>
<td>produced witnesses</td>
<td>3392</td>
</tr>
<tr>
<td>with invariant</td>
<td>1585</td>
</tr>
</tbody>
</table>

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

Invalid ACSL assertion for which no correct location exists
Evaluation - Results (3)

Validate generated ACSL annotations/witnesses

▶ Validation of produced ACSL annotations usually successful ✓

▶ Validation of produced witnesses succeeds almost always ✓

▶ No incorrect invariants after roundtrip ✓

<table>
<thead>
<tr>
<th>input programs</th>
<th>4685</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frama-C-SV true</td>
<td>3188</td>
</tr>
<tr>
<td>Frama-C-SV unknown</td>
<td>1445</td>
</tr>
<tr>
<td>Frama-C-SV other</td>
<td>52</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>input witnesses</th>
<th>3483</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>3463</td>
</tr>
<tr>
<td>ERROR (recursion)</td>
<td>19</td>
</tr>
<tr>
<td>TIMEOUT</td>
<td>1</td>
</tr>
</tbody>
</table>
Summary

- Conversion Witness Invariant \(\Rightarrow\) ACSL Annotation
  - Easy in theory: Just use invariant as predicate in ACSL assertion
  - Several approaches to find correct location for assertion
  - There might not be a correct location

- Conversion ACSL Annotation \(\Rightarrow\) Witness Invariant
  - Represent annotations by predicates
  - Predicates can then be converted to invariants
  - Bigger contracts can be split up into multiple assertions before conversion

- Future Work
  - Better way to extract invariants from witnesses
  - Improve parsing of ACSL annotations