

Jessie Plugin and ACSL Specifications

Virgile Prevosto

CEA List

March 31, 2009

(long no
[for i < 0
C1); if (0)
tmp2 =
st of the

tmp2[j] = (t < (Nb1 - 1)) ? else if (tmp1[j] >= (t < (Nb1 - 1)) ? tmp2[j] = (t < (Nb1 - 1)) ? 0; else tmp2[j] = tmp1[j]; /* Then the second pass looks like the first one: */
tmp1[0] = 0; k = 0; k++ tmp1[k] = mc2[0][k] * tmp2[k]; /* The [i][j] coefficient of the matrix product MC2*TMP2, that is: *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1 *MC1 *
i = 1; tmp1[0] >= 1; /* Final rounding: tmp2[0] is now represented on 9 bits. */ if (tmp1[0] < -255) m2[0] = -255; else if (tmp1[0] > 255) m2[0] = 255; else m2[0] = tmp1[0];



Jessie Usage

Function Contracts

Generating Proof Obligations

Advanced Specification

Example 1: Searching

Example 2: Sorting

(long n;
for (i = 0;
i < n; i++)
tmp2 =
... of the

tmp2[i] = (i < n/2) ? tmp[i] : tmp[n-i];
tmp1[i] = 0; k = 0; k++
tmp1[i] = mc2[i][k] * tmp2[k];
i = 1; tmp1[0] >= 1; Final rounding: tmp2[0] is now represented on 9 bits.
if (tmp1[0] < -256) tmp2[0] = -256; else if (tmp1[0] > 255) tmp2[0] = 255; else tmp2[0] = tmp1[0];
Then the second pass looks like the first one: *MC2*(TMP1) = *MC2*(MC1*M1) = *MC2*M1*MC1 = *MC1*
The [i,j] coefficient of the matrix product MC2*TMP2, that is, *MC2*[i](TMP1) = *MC2*[i](MC1*M1) = *MC2*M1*[i](MC1 =





- ▶ Hoare-logic based plugin, developed at INRIA Saclay.
- ▶ Input: a program and a specification
- ▶ Jessie generates **verification conditions**
- ▶ Use of **Automated Theorem Provers** to discharge the VCs
- ▶ If all VCs are proved, **the program is correct** with respect to the specification
- ▶ Otherwise: need to investigate why the proof fails
 - ▶ Fix bug in the code
 - ▶ Adds additional annotations to help ATP
 - ▶ Interactive Proof (Coq/Isabelle)



- ▶ Hoare-logic based plugin, developed at INRIA Saclay.
- ▶ Input: a program and a specification
- ▶ Jessie generates **verification conditions**
- ▶ Use of **Automated Theorem Provers** to discharge the VCs
- ▶ If all VCs are proved, **the program is correct** with respect to the specification
- ▶ Otherwise: need to investigate why the proof fails
 - ▶ Fix bug in the code
 - ▶ Adds additional annotations to help ATP
 - ▶ Interactive Proof (Coq/Isabelle)



- ▶ Hoare-logic based plugin, developed at INRIA Saclay.
- ▶ Input: a program and a specification
- ▶ Jessie generates **verification conditions**
- ▶ Use of **Automated Theorem Provers** to discharge the VCs
- ▶ If all VCs are proved, **the program is correct** with respect to the specification
- ▶ Otherwise: need to investigate why the proof fails
 - ▶ Fix bug in the code
 - ▶ Adds additional annotations to help ATP
 - ▶ Interactive Proof (Coq/Isabelle)



- ▶ Hoare-logic based plugin, developed at INRIA Saclay.
- ▶ Input: a program and a specification
- ▶ Jessie generates **verification conditions**
- ▶ Use of **Automated Theorem Provers** to discharge the VCs
- ▶ If all VCs are proved, **the program is correct** with respect to the specification
- ▶ Otherwise: need to investigate why the proof fails
 - ▶ Fix bug in the code
 - ▶ Adds additional annotations to help ATP
 - ▶ Interactive Proof (Coq/Isabelle)



- ▶ Hoare-logic based plugin, developed at INRIA Saclay.
- ▶ Input: a program and a specification
- ▶ Jessie generates **verification conditions**
- ▶ Use of **Automated Theorem Provers** to discharge the VCs
- ▶ If all VCs are proved, **the program is correct** with respect to the specification
- ▶ Otherwise: need to investigate why the proof fails
 - ▶ Fix bug in the code
 - ▶ Adds additional annotations to help ATP
 - ▶ Interactive Proof (Coq/Isabelle)



- ▶ Hoare-logic based plugin, developed at INRIA Saclay.
- ▶ Input: a program and a specification
- ▶ Jessie generates **verification conditions**
- ▶ Use of **Automated Theorem Provers** to discharge the VCs
- ▶ If all VCs are proved, **the program is correct** with respect to the specification
- ▶ Otherwise: need to investigate why the proof fails
 - ▶ Fix bug in the code
 - ▶ Adds additional annotations to help ATP
 - ▶ Interactive Proof (Coq/Isabelle)



- ▶ Hoare-logic based plugin, developed at INRIA Saclay.
- ▶ Input: a program and a specification
- ▶ Jessie generates **verification conditions**
- ▶ Use of **Automated Theorem Provers** to discharge the VCs
- ▶ If all VCs are proved, **the program is correct** with respect to the specification
- ▶ Otherwise: need to investigate why the proof fails
 - ▶ Fix bug in the code
 - ▶ Adds additional annotations to help ATP
 - ▶ Interactive Proof (Coq/Isabelle)



- ▶ Hoare-logic based plugin, developed at INRIA Saclay.
- ▶ Input: a program and a specification
- ▶ Jessie generates **verification conditions**
- ▶ Use of **Automated Theorem Provers** to discharge the VCs
- ▶ If all VCs are proved, **the program is correct** with respect to the specification
- ▶ Otherwise: need to investigate why the proof fails
 - ▶ Fix bug in the code
 - ▶ Adds additional annotations to help ATP
 - ▶ Interactive Proof (Coq/Isabelle)



- ▶ Hoare-logic based plugin, developed at INRIA Saclay.
- ▶ Input: a program and a specification
- ▶ Jessie generates **verification conditions**
- ▶ Use of **Automated Theorem Provers** to discharge the VCs
- ▶ If all VCs are proved, **the program is correct** with respect to the specification
- ▶ Otherwise: need to investigate why the proof fails
 - ▶ Fix bug in the code
 - ▶ Adds additional annotations to help ATP
 - ▶ Interactive Proof (Coq/Isabelle)



Usage

- ▶ Proof of functional properties of the program
- ▶ Modular verification (function per function)

Limitations

- ▶ Cast between pointers and integers
- ▶ Limited support for union type
- ▶ Aliasing requires some care

```
(long n)
{ for (i = 0; i < n; i++)
  tmp2[i] = 0;
  if (n > 0)
    tmp2[n-1] = 1;
  return tmp2;
}
```

```
tmp2[0] = 1; if (n < 0) tmp2[0] = 0; else if (tmp2[0] > 1) tmp2[0] = 1; else if (tmp2[0] < 0) tmp2[0] = 0; else tmp2[0] = tmp2[0]; /* Then the second pass looks like the first one: */
tmp2[1] = 0; k = 1; while (k < n) { tmp2[k] = mc2[k] * tmp2[k]; /* The [k] coefficient of the matrix product MC2*TMP2, that is, *MC2*[TMP1] = MC2*[MC1*TM1] = MC2*TM1*MC1
k = k + 1; tmp2[k] >= 1; /* Final rounding: tmp2[k] is now represented on 9 bits. */ if (tmp2[k] < -255) tmp2[k] = -255; else if (tmp2[k] > 255) tmp2[k] = 255; else tmp2[k] = tmp2[k];
return tmp2;
}
```



Usage

- ▶ Proof of functional properties of the program
- ▶ Modular verification (function per function)

Limitations

- ▶ Cast between pointers and integers
- ▶ Limited support for union type
- ▶ Aliasing requires some care

```
(long n)
{ for (i = 0; i < n; i++)
  C[i] = 0;
  tmp2 = ...
  // ...
}
```

```
tmp2[i] = (i < (N-1) ? tmp[i] : 0); else if (tmp1[i] >= 0) { if (i < (N-1) ? tmp2[i] : 0) else tmp2[i] = tmp1[i]; } // Then the second pass looks like the first one:
tmp1[i] = 0; k = 0; k++ tmp1[i] = mc2[i][k] * tmp2[k]; // The [i][k] coefficient of the matrix product MC2*TMP2, that is: *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1 * MC1
i = i + 1; tmp1[i] >= 0; } // Final rounding: tmp2[i] is now represented on 9 bits. *if (tmp1[i] < -256) tmp2[i] = -256; else if (tmp1[i] > 255) tmp2[i] = 255; else tmp2[i] = tmp1[i];
```



Usage

- ▶ Proof of functional properties of the program
- ▶ Modular verification (function per function)

Limitations

- ▶ Cast between pointers and integers
- ▶ Limited support for union type
- ▶ Aliasing requires some care

```
(long n)
{ for (i = 0; i < n; i++)
  C[i] = 0;
  tmp2 = ...
  // ...
}
```

```
tmp2[i] = (i < (N-1) ? tmp[i] : 0); else if (tmp1[i] >= 0) { if (i < (N-1) ? tmp2[i] : 0) else tmp2[i] = tmp1[i]; } // Then the second pass looks like the first one:
tmp1[i] = 0; k = 0; k++ tmp1[i] = mc2[i][k] * tmp2[k]; // The [i][k] coefficient of the matrix product MC2*TMP2, that is: *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1 * MC1
i = i + 1; tmp1[i] >= 0; } // Final rounding: tmp2[i] is now represented on 9 bits. *if (tmp1[i] < -256) tmp2[i] = -256; else if (tmp1[i] > 255) tmp2[i] = 255; else tmp2[i] = tmp1[i];
```



Usage

- ▶ Proof of functional properties of the program
- ▶ Modular verification (function per function)

Limitations

- ▶ Cast between pointers and integers
- ▶ Limited support for union type
- ▶ Aliasing requires some care



Usage

- ▶ Proof of functional properties of the program
- ▶ Modular verification (function per function)

Limitations

- ▶ Cast between pointers and integers
- ▶ Limited support for union type
- ▶ Aliasing requires some care



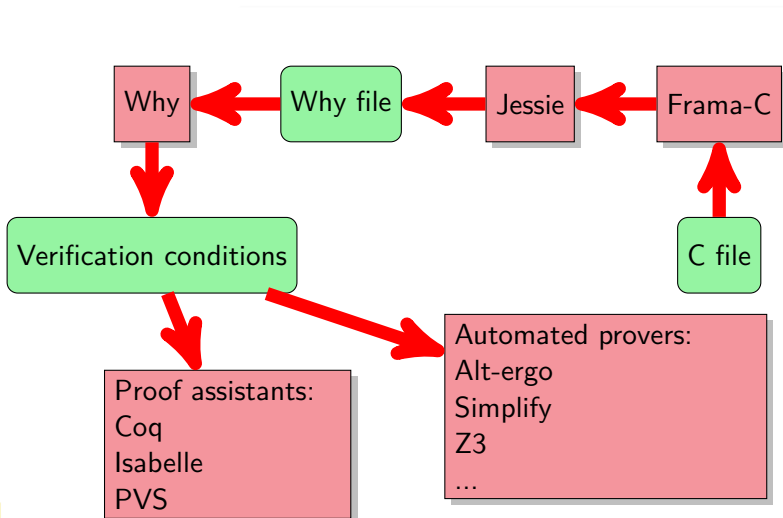
Usage

- ▶ Proof of functional properties of the program
- ▶ Modular verification (function per function)

Limitations

- ▶ Cast between pointers and integers
- ▶ Limited support for union type
- ▶ Aliasing requires some care





Check safety of a function

- ▶ Pointer accesses
- ▶ Arithmetic overflow
- ▶ Division

```
unsigned int M;
```

```
void mean(unsigned int* p, unsigned int* q) {
    M = (*p + *q) / 2;
}
```



Jessie Usage

Function Contracts

Generating Proof Obligations

Advanced Specification

Example 1: Searching

Example 2: Sorting

```
(long n)
{ for (i = 0; i < n; i++)
  C1: if (0)
    tmp2 = ...
  // ...
}
```

```
tmp2[i] = (i < (N-1) ? tmp1[i] : 0); // Then the second pass looks like the first one:
tmp1[0] = 0; k = 0; k++ tmp1[k] = mc2[k] * tmp2[k]; // The [k] coefficient of the matrix product MC2*TMP2, that is: *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1*MC1
i = 1; tmp1[0] >= 1; // Final rounding: tmp2[0] is now represented on 9 bits. *if (tmp1[0] < -256) m2[0] = -256; else if (tmp1[0] > 255) m2[0] = 255; else m2[0] = tmp1[0];
```



- ▶ Functional specification
- ▶ Pre-conditions (requires)
- ▶ Post-conditions (ensures)

Example

```
unsigned int M;
```

```
/*@
```

```
  requires \valid(p) & \valid(q);
```

```
  ensures M ≡ (*p + *q) / 2;
```

```
*/
```

```
void mean(unsigned int* p, unsigned int* q) {
```

```
  if (*p ≥ *q) { M = (*p - *q) / 2 + *q; }
```

```
  else { M = (*q - *p) / 2 + *p; }
```

```
}
```



- ▶ Functional specification
- ▶ Pre-conditions (requires)
- ▶ Post-conditions (ensures)

Example

```
unsigned int M;
```

```
/*@
```

```
  requires \valid(p) ^ \valid(q);
```

```
  ensures M ≡ (*p + *q) / 2;
```

```
*/
```

```
void mean(unsigned int* p, unsigned int* q) {
```

```
  if (*p ≥ *q) { M = (*p - *q) / 2 + *q; }
```

```
  else { M = (*q - *p) / 2 + *p; }
```

```
}
```



- ▶ Functional specification
- ▶ Pre-conditions (requires)
- ▶ Post-conditions (ensures)

Example

```
unsigned int M;
```

```
/*@
```

```
  requires \valid(p) ^ \valid(q);
```

```
  ensures M ≡ (*p + *q) / 2;
```

```
*/
```

```
void mean(unsigned int* p, unsigned int* q) {
```

```
  if (*p ≥ *q) { M = (*p - *q) / 2 + *q; }
```

```
  else { M = (*q - *p) / 2 + *p; }
```

```
}
```



The specification:

```
/*@
```

```
  requires \valid(p) ∧ \valid(q);
```

```
  ensures M ≡ (*p + *q) / 2;
```

```
  assigns M;
```

```
*/
```

```
void mean(unsigned int* p, unsigned int* q);
```



Jessie Usage

Function Contracts

Generating Proof Obligations

Advanced Specification

Example 1: Searching

Example 2: Sorting

```
(long no
[ for i = 0
C1) if (0)
tmp2 =
st of the
```

```
tmp2[0] = (t <= (Nb1 - 1)) else if (tmp1[0]) >= (t <= (Nb1 - 1)) tmp2[0] = (t <= (Nb1 - 1)) - 1; else tmp2[0] = tmp1[0]; /* Then the second pass looks like the first one: */ Nb1
tmp1[0] = 0; k = 0; k++ tmp1[0][k] = mc2[0][k] * tmp2[0][k]; /* The [i][j] coefficient of the matrix product MC2*TMP2, that is, *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1*MC1
i = 1; tmp1[0][i] >= 1; /* Final rounding: tmp2[0][i] is now represented on 9 bits. */ if (tmp1[0][i] < -256) m2[0][i] = -256; else if (tmp1[0][i] > 255) m2[0][i] = 255; else m2[0][i] = tmp1[0][i]; /*
```



- ▶ Introduced by Floyd and Hoare (70s)
- ▶ Hoare triple: $\{P\}s\{Q\}$, meaning: *If P holds, then Q will hold after the execution of statement s*
- ▶ Deduction rules on Hoare triples: *Axiomatic semantic*



$$\frac{\overline{\{P\}\{\}\{P\}}}{\{P\} \Rightarrow P'} \quad \frac{\{P'\} \text{ s } \{Q'\} \quad Q' \Rightarrow Q}{\{P\} \text{ s } \{Q\}}$$

$$\frac{\{P\} \text{ s_1 } \{R\} \quad \{R\} \text{ s_2 } \{Q\}}{\{P\} \text{ s_1 ; s_2 } \{Q\}} \quad \frac{e \text{ evaluates without error}}{\{P[x \leftarrow e]\} \text{ x=e ; } \{P\}}$$

$$\frac{\{P \wedge e\} \text{ s_1 } \{Q\} \quad \{P \wedge !e\} \text{ s_2 } \{Q\}}{\{P\} \text{ if } (e) \{ \text{ s_1 } \} \text{ else } \{ \text{ s_2 } \} \{Q\}}$$

$$\frac{\{I \wedge e\} \text{ s } \{I\}}{\{I\} \text{ while } (e) \{ \text{ s } \} \{I \wedge !e\}}$$



- ▶ Program seen as a **predicate transformer**
- ▶ Given a function s , a pre-condition Pre and a post-condition $Post$
- ▶ We start from $Post$ at the end of the function and go backwards
- ▶ At each step, we have a property Q and a statement s , and compute the *weakest pre-condition* P such that $\{P\}s\{Q\}$ is a valid Hoare triple.
- ▶ When we reach the beginning of the function with property P , we must prove $Pre \Rightarrow P$.



- ▶ Program seen as a **predicate transformer**
- ▶ Given a function s , a pre-condition Pre and a post-condition $Post$
- ▶ We start from $Post$ at the end of the function and go backwards
- ▶ At each step, we have a property Q and a statement s , and compute the *weakest pre-condition* P such that $\{P\}s\{Q\}$ is a valid Hoare triple.
- ▶ When we reach the beginning of the function with property P , we must prove $Pre \Rightarrow P$.



- ▶ Program seen as a **predicate transformer**
- ▶ Given a function s , a pre-condition Pre and a post-condition $Post$
- ▶ We start from $Post$ at the end of the function and go backwards
- ▶ At each step, we have a property Q and a statement s , and compute the *weakest pre-condition* P such that $\{P\}s\{Q\}$ is a valid Hoare triple.
- ▶ When we reach the beginning of the function with property P , we must prove $Pre \Rightarrow P$.



- ▶ Program seen as a **predicate transformer**
- ▶ Given a function s , a pre-condition Pre and a post-condition $Post$
- ▶ We start from $Post$ at the end of the function and go backwards
- ▶ At each step, we have a property Q and a statement s , and compute the *weakest pre-condition* P such that $\{P\}s\{Q\}$ is a valid Hoare triple.
- ▶ When we reach the beginning of the function with property P , we must prove $Pre \Rightarrow P$.



- ▶ Program seen as a **predicate transformer**
- ▶ Given a function s , a pre-condition Pre and a post-condition $Post$
- ▶ We start from $Post$ at the end of the function and go backwards
- ▶ At each step, we have a property Q and a statement s , and compute the *weakest pre-condition* P such that $\{P\}s\{Q\}$ is a valid Hoare triple.
- ▶ When we reach the beginning of the function with property P , we must prove $Pre \Rightarrow P$.



► Assignment

$$WP(x=e, Q) = Q[x \leftarrow e]$$

► Sequence

$$WP(s_1; s_2, Q) = WP(s_1, WP(s_2, Q))$$

► Conditional

$$WP(\text{if } (e) \{ s_1 \} \text{ else } \{ s_2 \}, Q) = e \Rightarrow WP(s_1, Q) \wedge !e \Rightarrow WP(s_2, Q)$$

► While

$$WP(\text{while } (e) \{ s \}, Q) = I \wedge \forall \omega. I \Rightarrow (e \Rightarrow WP(s, I) \wedge !e \Rightarrow Q)$$



Issue

How can we represent memory operations ($*x$, $a[i]=42, \dots$) in the logic

- ▶ If too low-level (a big array of bytes), proof obligations are intractable.
- ▶ If too abstract, some C constructions can not be represented (arbitrary pointer casts, aliasing)
- ▶ Standard solution (Burstal-Bornat): replace struct's components by a function



Issue

The same memory location can be accessed through different means:

```
int y;
int* yptr = &y;
*yptr = 3;
/*@ assert y == 3; */
```

- ▶ Again, supposing that any two pointers can be aliases would lead to intractable proof obligations.
- ▶ Memory is separated in disjoint regions
- ▶ Some hypotheses are done (as additional pre-conditions)



Jessie Usage

Function Contracts

Generating Proof Obligations

Advanced Specification

Example 1: Searching

Example 2: Sorting

```
(long n)
{ for (i = 0; i < n; i++)
  C1: if (0)
    tmp2 = ...
  // ...
}
```

```
tmp2[i] = (i < (N-1) ? tmp[i] : 0); // Then the second pass looks like the first one:
tmp1[0] = 0; k = 0; k++ tmp1[k] = mc2[i][k] * tmp2[k]; // The [i][k] coefficient of the matrix product MC2*TMP2, that is: *MC2*(TMP1) = MC2*(MC1*M1) = MC2*M1*MC1
i = 1; tmp1[0] >= 1; // Final rounding: tmp2[0] is now represented on 9 bits. *if (tmp1[0] < -256) m2[0] = -256; else if (tmp1[0] > 255) m2[0] = 255; else m2[0] = tmp1[0];
```



Informal spec

- Input: a **sorted** array and its length, an element to search.
- Output: index of the element or -1 if not found

Implementation

```
int find_array(int* arr, int length, int query) {
    int low = 0;
    int high = length - 1;
    while (low ≤ high) {
        int mean = low + (high - low) / 2;
        if (arr[mean] ≡ query) return mean;
        if (arr[mean] < query) low = mean + 1;
        else high = mean - 1;
    }
    return -1;
}
```



Informal specification

- ▶ Input: an array and its length
- ▶ Output: the array is sorted in ascending order

```
int index_min(int* a, int low, int high);
```

```
void swap(int* arr, int i, int j);
```

```
void min_sort(int* arr, int length) {
    for(int i = 0; i < length; i++) {
        int min = index_min(arr,i,length);
        swap(arr,i,min);
    }
}
```

