A Trustworthy Framework for Resource-Aware Embedded Programming

Christopher Brown
University of St Andrews

Teamplay Workshop
May 2021
Functional requirements
What the program does/computes

Non-functional requirements
• Worst-Case Execution Time
• Memory/storage limitations
• Energy budget
• Security

STM32*-Discovery 32-bit ARM Cortex-M4,
192-Kbyte RAM,
1-Mbyte Flash
Functional requirements

What the program does/computes

STM32*-Discovery 32-bit ARM Cortex-M4,
192-Kbyte RAM,
1-Mbyte Flash
Functional requirements
What the program does/computes

Non-functional requirements
- Worst-Case Execution Time
- Memory/storage limitations
- Energy budget
- Security

STM32*-Discovery 32-bit ARM Cortex-M4, 192-Kbyte RAM, 1-Mbyte Flash
### Functional Requirements

- Inspecting output
- Testing
- Proofs of functionality
Adhering to Requirements

**Functional Requirements**
- Inspecting output
- Testing
- Proofs of functionality

**Non-Functional Requirements**
- Analyse/profile code (e.g. WCET analysis, energy consumption)
- Compare whether results of analysis/profiling fit within requirements
Analysing and Measuring for Information

- Lots of extrinsic tools
  - CoMET, aiT
  - Maximum-Stack-Depth analysers
  - Instruction Set Analysis
  - Hardware Performance Counters
  - Structural Attack Model
Analysing and Measuring for Information

- **Lots** of extrinsic tools
  - CoMET, aiT
  - Maximum-Stack-Depth analysers
  - Instruction Set Analysis
  - Hardware Performance Counters
  - Structural Attack Model

- Generally **no single point of access/reference**
  - Different models, requirements
  - Different level of operation (source, instruction, &c.)
• **Lots** of extrinsic tools
  ◦ CoMET, aiT
  ◦ Maximum-Stack-Depth analysers
  ◦ Instruction Set Analysis
  ◦ Hardware Performance Counters
  ◦ Structural Attack Model

• Generally **no single point of access/reference**
  ◦ Different models, requirements
  ◦ Different level of operation (source, instruction, &c.)

• **Different levels of expertise required**
  ◦ Additional burdens the domain expert programmer
A Nicer Approach

- **Single high-level interface** to capture non-functional information
- Non-functional information **automatically captured** during compilation/runtime
  - Static & dynamic measurements provided by extrinsic tools
Contract Specification Language (CSL)

- Embedded Domain Specific Language for C
- `#include <csl.h>`
- **Annotate** C statements to capture non-functional information
  - Each annotates the following statement
  - Non-functional information is stored in a variable passed to the annotation
- Each annotation corresponds to an analysis/measurement

**Example Annotations**

- `__csl_time_worst(&var);`
- `__csl_energy_best(&var);`
- `__csl_time_average_iter(&var);`
- `__csl_profile_time_worst(&var);`
- `__csl_security_power_sca(&var);`
Example: Dijkstra

```c
int dijkstra(int chStart, int chEnd) {
    ... // set-up
    if (chStart == chEnd) { return 0; } else {
        ... // more set-up

        enqueue (chStart, 0, NONE); // starting from node chStart

        while (qcount() > 0) {
            QITEM *tmp = dequeue (&iNode, &iDist, &iPrev);
            ... // memory management
            for (i = 0; i < NUM_NODES; i++) {
                iCost = AdjMatrix[iNode][i]; // get cost of current node to i
                if (iCost != NONE) {
                    if ((NONE == rgnNodes[i].iDist) ||
                        (rgnNodes[i].iDist > (iCost + iDist))) {
                        // recorded distance in output matrix is empty/greater
                        rgnNodes[i].iDist = iDist + iCost;
                        rgnNodes[i].iPrev = iNode;
                        enqueue (i, iDist + iCost, iNode);
                    }
                }
            }
            ... // more braces...

            return rgnNodes[chEnd].iDist;
        }
    }
}
```
```c
int dijkstra(int chStart, int chEnd) {
    ... // set-up
    if (chStart == chEnd) { return 0; } else {
        ... // more set-up
    
    enqueue (chStart, 0, NONE); // starting from node chStart

    while (qcount() > 0) {
        QITEM *tmp = dequeue (&iNode, &iDist, &iPrev);
        ... // memory management
        __csl_time_worst_iter(&selection_time);
        for (i = 0; i < NUM_NODES; i++) {
            iCost = AdjMatrix[iNode][i]; // get cost of current node to i
            if (iCost != NONE) {
                if ((NONE == rgnNodes[i].iDist) ||
                    (rgnNodes[i].iDist > (iCost + iDist))) {
                    ... // more braces...
            }
            ... // more braces...
        }
        return rgnNodes[chEnd].iDist;
    }
    ... 

    __csl_time_worst(&time_spent);
    output[output_count] = dijkstra(i,j);
```
First-Class Non-Functional Information

- Express captured non-functional information as **normal program values**
- Captured non-functional information becomes a **first-class citizen**
- Can be used to make decisions in the code
  - e.g. actively select images to send back to base or simply store all images depending on battery life of a drone
- Facilitates **contracts** (assertions)
  - Express non-functional requirements *in the code*
### Assertion Expression Language

| `a_1`, `a_2` : `AExp` | ::= | `\mathbb{N}` | `\times` | `a_1 + a_2` | `a_1 \times a_2` |
| | | | | `a_1 - a_2` | `a_1 \div a_2` | `\log_2 a_1` |
| `b_1`, `b_2` : `BExp` | ::= | `b_1 \land b_2` | `b_1 \lor b_2` | `\neg b_1` | `a_1 = a_2` | `a_1 \leq a_2` |

### Caveats

- All variables are numerical
- Non-natural numbers (in C) are assumed to be transformed uniformly
- Assertion expressions can also contain *non-ground* variables (we’ll come back to this a bit later...)
Idris and Dependent Types

‘Pure functional programming language with dependent types’

- We define types to express properties that we’re interested in
  - $x$ is a member of some vector $xs$
- Inhabitants of types are proofs.
- The type system acts as a theorem prover... the program is the proof object.
- Relies on the Curry-Howard Correspondence, where proofs can be represented as programs.
Idris: Example

1 data Vect : Nat -> Type where
2   [] : Vect 0 ty
3   (::) : (x : ty) -> (xs : Vect k ty) -> Vect (k + 1) ty

• Here the length of the list is encoded inside the type:

1   > [1,2,3]
2   [1,2, 3] : Vect 3 Integer
map : (f : a -> b) -> (xs : Vect k a) -> Vect k b

• This allows us to apply a function to every element of a list, without changing the length of the list:

> map (\x => x + 1) [1,2,3]
[2, 3, 4] : Vect 3 Integer

• The type system is stronger in fact: the definition of map encodes a proof that the implementation cannot change its length.
We can define equality \((x = y)\) by a reflexive type.

If \(x\) and \(y\) are the same members of a set, they are equal.

We can prove this by the function \(\text{decEq}\) which returns a proof object, \(\text{Refl}\), that \([1,2,3]\) is reflexive (i.e. equal to itself):

\[
\text{decEq : (xs : Vect k ty) -> (ys : Vect k ty) -> Dec (xs = ys)}
\]

\[
\begin{align*}
1 & \quad \text{decEq \ [1, 2, 3] \ [1, 2, 3]} \\
2 & \quad \text{Yes Refl : Dec ([1, 2, 3] = [1, 2, 3])}
\end{align*}
\]
The Process for Proving Assertions

- **Idris Model**: equivalent representation of the given annotated C program
- **Verified Idris Model**: assertions are extended with their proofs
The Process for Proving Assertions

- **Idris Model**: equivalent representation of the given annotated C program
- **Verified Idris Model**: assertions are extended with their proofs
- **Context Inference**: determines the values for variables in assertions
- **Well-Formedness**:
  - No out-of-bound array accesses
  - All variables are instantiated before use;
  - Loops are unrolled to enable generation of all relevant proofs of any assertions loop body
Proving Assertions using Idris

```haskell
__csl_assert(time_spent <= NUM_NODES * NUM_NODES * selection_time);
```
Proving Assertions using Idris

```idris
__csl_assert(time_spent <= NUM_NODES * NUM_NODES * selection_time);

dijkstra_assert : Env -> Assertion
dijkstra_assert env =
  let
    p0  = (Var "time_spent")
    p0' = eval env p0
    p1  = (Var "NUM_NODES")
    p1' = eval env p1
    p2  = (Var "comparison_time")
    p2' = eval env p2
    p3  = (Mul (Mul p1 p1) p2)
    p3' = eval env p3
  in
    MkAssertion ((LTE p0 p3)
      (MkEvald p0 p0')
      (MkEvald p3 p3')
      (isLTE p0' p3'))
```
__csl_assert(time_spent <= NUM_NODES * NUM_NODES * selection_time);

1   dijkstra_assert : Env -> Assertion
2   dijkstra_assert env =
3       let
4           p0  = (Var "time_spent")
5           p0' = eval env p0
6
7           p1  = (Var "NUM_NODES")
8           p1' = eval env p1
9
10          p2  = (Var "comparison_time")
11          p2' = eval env p2
12
13          p3  = (Mul (Mul p1 p1) p2)
14          p3' = eval env p3
15
16       in
17       MkAssertion ((LTE p0 p3)
18                                    (MkEvald p0 p0')
19                                    (MkEvald p3 p3')
20                                    (isLTE p0' p3'))

MkAssertion (LTE (Var "time_spent")
(Mul (Mul (Var "NUM_NODES") (Var "NUM_NODES"))
 (Var "comparison_time")))
(MkEvald (Var "time_spent") 32933)
(MkEvald ((Mul (Mul (Var "NUM_NODES")
 (Var "NUM_NODES"))) (Var "comparison_time"))) 1507328)
(Yes (LTESucc (...)))
Problems with the Current Approach

1. Abstraction and context generation is not shown to be sound
   ◦ If we are going to give proofs, they need to be meaningful
2. Abstraction is not very well defined
   ◦ A small collection of functions that produce an unintelligible proof
   ◦ Does not make full use of the dependent types
     ● E.g. resulting proof is not indexed by program or context
3. Literal values are strictly natural numbers
   ◦ C supports a range of number types
4. Can only deal with ground terms
   ◦ E.g. what happens if we cannot evaluate a sub-term of the assertion to a normal form?
Lang: a subset of C
Zipper: facilitates context inference

Abstract Interpretation: all assertions have proofs of their holding true or false
A Subset of C

\[a, a_1, a_2 : AExp ::= n \mid x \mid a_1 + a_2\]
\[b : BExp ::= a_1 = a_2\]
\[s, k : Stmt ::= x := a; s \mid \text{stop}\]
\[p : AStmt ::= \text{cert } b \ p \mid \text{pure } s \ p \mid \text{halt}\]

\[n : \mathbb{N}, \quad x : \text{Var}\]
A Subset of C

\[
\begin{align*}
  a, a_1, a_2 : \text{AExp} &::= n \mid x \mid a_1 + a_2 \\
  b : \text{BExp} &::= a_1 = a_2 \\
  s, k : \text{Stmt} &::= x := a; s \mid \text{while } b s k \mid \text{stop} \\
  p : \text{AStmt} &::= \text{cert } b p \mid \text{pure } s p \mid \text{halt}
\end{align*}
\]

\[
n : \mathbb{N}, \quad x : \text{Var}
\]

There are many other constructs, e.g. loops, arrays, etc. but we omit them here for simplicity.
A Subset of C

\[ a, a_1, a_2 : \text{AExp} \quad ::= \quad n \mid x \mid a_1 + a_2 \]
\[ b : \text{BExp} \quad ::= \quad a_1 = a_2 \]
\[ s, k : \text{Stmt} \quad ::= \quad x := a; s \mid \text{while } b s k \mid \text{stop} \]
\[ p : \text{AStmt} \quad ::= \quad \text{cert } b p \mid \text{pure } s p \mid \text{halt} \]

\[ n : \mathbb{N}, \quad x : \text{Var} \]

There are many other constructs, e.g. loops, arrays, etc. but we omit them here for simplicity.

For our purposes, capture annotations are semantically equivalent to assignment statements and will be added later.
A Subset of C

\[ a, a_1, a_2 : \text{AEexp} \ ::= \ n \mid x \mid a_1 + a_2 \]
\[ b : \text{Bexp} \ ::= \ a_1 = a_2 \]
\[ s, k : \text{Stmt} \ ::= \ x := a; s \mid \text{while} \ b \ s \ k \mid \text{stop} \]
\[ p : \text{AStmt} \ ::= \ \text{cert} \ b \ p \mid \text{pure} \ s \ p \mid \text{halt} \]

\[ n : \mathbb{N}, \quad x : \text{Var} \]

There are many other constructs, e.g. loops, arrays, etc. but we omit them here for simplicity.

For our purposes, capture annotations are semantically equivalent to assignment statements and will be added later.

\[ \text{Var} : \text{Type} \]
\[ \text{Var} = \text{Elem} \ \text{NumVar} \ (\text{Vect} \ n \ (\text{Variable} \ \text{Numeric})) \]
Well-Formed Programs

1. Well-typed
2. Variables assigned before use
Well-Formed Programs

Let $\text{Env} : (\text{vs} : \langle \text{Vect nvs} \ (\text{Variable Numerical}) \rangle) \rightarrow \text{Type}$ be defined as $\text{Env vs} = \langle \text{Vect n (Elem NumVar vs)} \rangle$.

**Data WFAExp**: $(a : \text{AExp vs}) \rightarrow (\text{env} : \text{Env vs}) \rightarrow \text{Type} \quad \text{where}

- $\text{WFVal} : \text{WFAExp (Val n) env}$
- $\text{WFVar} : (\text{wfprf} : \langle \text{Elem var env} \rangle) \rightarrow \text{WFAExp (Var var) env}$
- $\text{WFAdd} : (\text{wfa1} : \text{WFAExp a1 env}) \rightarrow (\text{wfa2} : \text{WFAExp a2 env}) \rightarrow \text{WFAExp (Add a1 a2) env}$

**IsWFAExp**: $(a : \text{AExp vs}) \rightarrow (\text{env} : \text{Env vs}) \rightarrow \text{Dec (WFAExp a env)}$

**Data WFBExp**: $(b : \text{BExp vs}) \rightarrow (\text{env} : \text{Env vs}) \rightarrow \text{Type} \quad \text{where}

- $\text{WFEq} : (\text{wfa1} : \text{WFAExp a1 env}) \rightarrow (\text{wfa2} : \text{WFAExp a2 env}) \rightarrow \text{WFBExp (Eq a1 a2) env}$

**IsWFBExp**: $(b : \text{BExp vs}) \rightarrow (\text{env} : \text{Env vs}) \rightarrow \text{Dec (WFBExp b env)}$

**Data WFStmt**: $(s : \text{Stmt vs}) \rightarrow (\text{env} : \text{Env vs}) \rightarrow \text{Type} \quad \text{where}

- $\text{WFVAsn} : (\text{wfa} : \text{WFAExp a1 env}) \rightarrow (\text{wfk} : \text{WFStmt k (var::env)}) \rightarrow \text{WFStmt (VAsn var a1 k) env}$
- $\text{WFStop} : \text{WFStmt Stop env}$

**IsWFStmt**: $(s : \text{Stmt vs}) \rightarrow (\text{env} : \text{Env vs}) \rightarrow \text{Dec (WFStmt s env)}$
Well-Formed Programs

25 Env : (vs : Vect nvs (Variable Numerical)) -> Type
26 Env vs = Vect n (Elem NumVar vs)
27
28 data WFAStmt : (as : AStmt vs) -> (env : Env vs) -> Type where
29      WFCert : (wfb : WFBExp b env) -> (wfak : WFAStmt ak env)
30      -> WFAStmt (Cert b ak) env
31      WFPure : (wfs : WFStmt (VAsn var a1 Stop) env)
32      -> (wfak : WFAStmt ak (var::env))
33      -> WFAStmt (Pure (VAsn var a1 Stop) ak) env
34      WFHalt : WFAStmt Halt env
35
36 isWFAStmt : (as : AStmt vs) -> (env : Env vs) -> Dec (WFAStmt as dvs)
Abstract Interpretation

- Brings everything together.
- Extends a well-formed program with a proof for each assertion contained therein
Abstract Interpretation

1  data VerifCert : \{env_0 : Env vs\} -- env at start of program
2     -> \{env_i : Env vs\} -- env at 'current' statement
3     -> (wfp : WFAStmt p env_0) -- program is well-formed
4     -> (wfprefix : WFAStmt prefix env_0) -- context is well-formed
5     -> (wfas : WFAStmt as env_i) -- 'current' stmt is well-formed
6     -> (zip : ZipperAStmt p prefix as) -- relates prefix and as
7     -> Type where
8
9  verifCert : (env_0 : Elem vs)
10     -> \{env_i : Elem vs\}
11     -> (prefix : AStmt vs)
12     -> (as : AStmt vs)
13     -> (wfp : WFAStmt (prefix ++ as) env_0)
14     -> (wfprefix : WFAStmt prefix env_0)
15     -> (wfas : WFAStmt as env_i)
16     -> (zip : ZipperAStmt (prefix ++ as) prefix as)
17     -> (Dec (VerifCert wfp wfprefix wfas zip))
Beyond Natural Numbers

- C supports different number types, and embedded systems may use those
  - We cannot assume that they will only use non-negative integers (i.e. natural numbers)
- Our framework should support generic carrier types
  - With guarantees about operations over specific carrier types
- Generalised carrier types will facilitate non-ground assertions
• Inspired by and using Slama & Brady’s *Automatically Proving Equivalence by Type-Safe Reflection* (2017)

• Extend language with carrier type argument

```haskell
data AExp : {c : Type} -> (vs : Vect nvs (Variable Numerical)) -> Type where
  Val : (n : c) -> AExp vs
  Var : (var : Elem NumVar vs) -> AExp vs
  Add : (a1 : AExp vs) -> (a2 : AExp vs) -> AExp vs
```
Beyond Natural Numbers

- Guarantees are made about \( c \) by a *hierarchy of interfaces*
- This hierarchy reflects *algebraic structures*, defining operations on \( c \)

```plaintext
interface Setoid c where
  equiv : c -> c -> Type
  refl : (x : c) -> equiv x x
  sym : {x, y : c} -> equiv x y -> equiv y x
  trans : {x, y, z : c} -> equiv x y -> equiv y z -> equiv x z
  set_equiv : (x : c) -> (y : c) -> Dec (equiv x y)

interface Setoid c => Magma c where
  (+) : c -> c -> c

interface Magma c => SemiGroup c where
  plus_assoc : (c1, c2, c3 : c) -> equiv ((c1 + c2) + c3) (c1 + (c2 + c3))
```
• **Generic**: does not rely upon a specific numeric type

• **Extensible**: new algebraic structures can be added

• Operations/properties are defined by the algebraic structure used

• **Only** operations supported by the selected algebraic structure can be used
Example: Summing a List

```c
int sumList(const int *xs) {
    int sum = 0;
    int loop_energy;
    __csl_energy(&loop_energy);
    for (i = 0; i < len(xs); i++){
        sum = sum + xs[i];
    }
    __csl_assert(sum == 15);
    __csl_assert(loop_energy < 10);
    return sum;
}
```
Example: Summing a List

```c
int sumList(const int *xs) {
    int sum = 0;
    int loop_energy;
    __csl_energy(&loop_energy);
    for (i = 0; i < len(xs); i++){
        sum = sum + xs[i];
    }
    __csl_assert(sum == 15);
    __csl_assert(loop_energy < 10);
    return sum;
}
```

sumListNatOSg : Stmt Nat T.orderedSemigroup T.varSet
sumListNatOSg =
{- Assignment of specific elements of xs omitted -}
Comp (Assn Here (Val 0)) $
Comp (Assn (There (There Here)) (Val 7)) $
Comp (Iter (There Here) Here Here 5 MkNotZero
    (Assn Here (Add (Var Here) (Var (There Here))))) $
Comp (Cert (Eq (Var Here) (Val 15))) $
Cert (LTE (Var (There (There Here))) (Val 10) OrdSemigroup)
Towards Non-Ground Assertions

• Define type-safe rewrites and a similarly type-safe rewriting system
• Use rewriting system to solve \((\text{in})\text{equations}\)
• Again, inspired by Slama & Brady’s \textit{Type-Safe Reflection}
  ◦ Use standard mathematical rules to derive answers to \((\text{in})\text{equations}\)
Non-Ground Assertions

```c
... 
unsigned long long NumberOfFrames = 0;
unsigned long long NumberOfFrames_Threshold = 5;
struct __csl_nonground_int frame_threshold_range;
unsigned long long total_battery = 100;
unsigned long long image_energy_cost = 0;
unsigned long long energy_allowed = 0.8 * total_battery;
...

if (Edge_tot[0]>Thresh_Edge_tot) {
    __csl_energy(&image_energy_cost);
    Send_Image();
}
NumberOfFrames++;

__csl_assert_nonground(&frame_threshold_range, image_energy_cost * 
    frame_threshold_range.minimum <= energy_allowed);

if (NumberOfFrames > frame_threshold_range.maximum)
    Skip_Frame();
... 
```
• Assertions may include a variable whose value is **not known**

• Determine the **lower-bound, upper-bound**, or **range** for which that value makes the assertion hold true
Using SMT Solvers for Non-Ground Assertions

C/CSSL parser (pycparser-based) → Logical formulator (Python) → SMT solver

- Satisfiability check
- Range model
- Optimal model
- Unsat core
- Proof

API
SMT-LIB
Non-Ground Certificate

```
1   Just (Success (NonGround (LTE (Mul (Var "image_energy_cost" TyInt)
2   (Var "frame_threshold_range" TyInt))
3   (Var "energy_allowed" TyInt))
4   "frame_threshold_range"
5   1
6   4)
7   (MkCtx ["image_energy_cost", Just "20", TyInt),
8   "frame_threshold_range", Nothing, TyInt),
9   "energy_allowed", Just "80", TyInt)])
10  (TmLTE (TmMul (TmVar (MkString [...])))
11  (TmVar (MkString [...])))
12  (TmVar (MkString [...]'))
13  (MkCtx [(MkString [...]'],
14   Just (Pos 20)),
15   (MkString [...]',
16   Nothing),
17   (MkString [...]', Just (Pos 80))]
18  IsValidCtx)
19  (WfLTE (WfMul RiMul (WfVar (IsElemCtx Here)) (WfVar (IsElemCtx (There
20   Here)))))
21  (WfVar (IsElemCtx (There (There Here)))))
22  (True (TrLTE (Pos 40)
23  (Pos 80)
Summary

- Well-defined and sound framework for capturing and reasoning about non-functional properties in programs for embedded systems
  - The Contract Specification Language
  - C library providing capture annotations and assertions
  - Complete context inference and proof generation for assertions
  - Extend framework to support while-statements
    - Update concomitant definitions and covering functions/decision procedures
  - Add capture annotations.
- Generalise framework over carrier type
  - Support beyond natural numbers
- Support for non-ground assertions
  - Type-safe automatic (in)equation solving.
Thank you!