Modular Extraction of Lustre Models from C Code

The Frama-C/Synchrone Plugin

Loïc Correnson Christophe Junke Fabien Siron

Université Paris-Saclay, CEA, List

1. Background

General Context

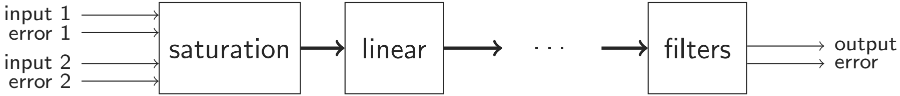
In the context of an ASNR [1] collaboration, we want to provide a tool to help analyzing the code of nuclear control systems.

Those systems are **synchronous-reactive programs** considered as being written in C:

1. No assumption on the language used to model/specify the system.
   - Part of the code can be automatically generated to C by some internal tools.
   - Part of the code can be manually written in C.
2. No assumption on the tool correctness used to (partially) generate the code.
   - For the generated code, the code generator cannot be considered as trusted.

[1] French Nuclear Safety and Radiation Protection Authority

Motivating Example



- Typically a cascade of nodes containing controllers and filters.
- Each node has parameters set during initialization.
- Each flow has both a value and an error flag, encoded by a bitfield.
- Typical properties to validate:
  - "*The output is included in the range [min-ϵ, max+ϵ].*"
  - "*If one of the inputs has an error bit set, then the output shall have an error bit set.*"
  - "*If no error bit is set, then the output behaves like its specification.*"

Frama-C

This project is done in the context of the Frama-C platform, a framework dedicated to the analysis of C code. It contains :

- **Frama-C/EVA**: abstract interpretation of C code
  - Can infer data invariants (e.g., value intervals)
  - Can infer memory invariants (e.g., pointer aliases)
  - Give an over-approximation of all cycles
- **Frama-C/WP**: deductive verification of C code
  - Hoare-style function contracts
  - Can prove properties on an individual function.
  - Not adapted to prove temporal properties.

Frama-C/Synchrone

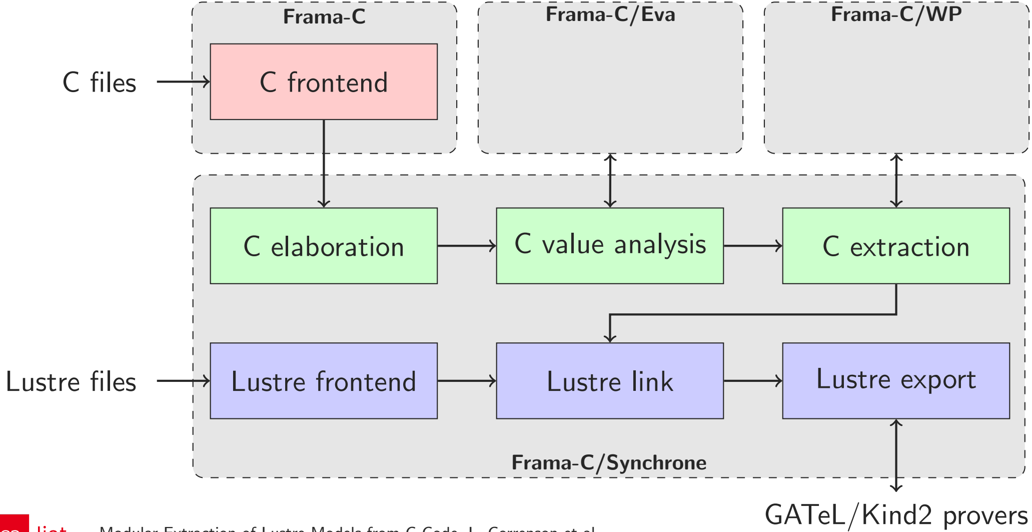
Proposition

Dedicated FRAMA-C plugin[a] , called FRAMA-C/SYNCHRONE:

- Based on FRAMA-C/EVA and FRAMA-C/WP to perform a modular extraction of LUSTRE from C code
- Based on GATeL and KIND2 as backends to verify proof obligations.

[a] B Blanc et al. 'Proving Properties of Reactive Programs From C to Lustre'. In: *ERTS 2018*. 2018.

Frama-C/Synchrone



2. Modular Extraction of Lustre programs

Modular Extraction: global approach

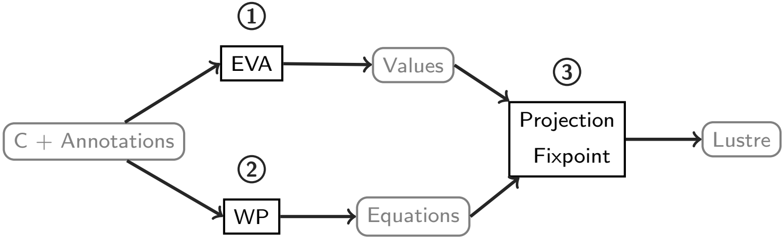


Figure: Extraction Methodology

Modular Extraction: global approach

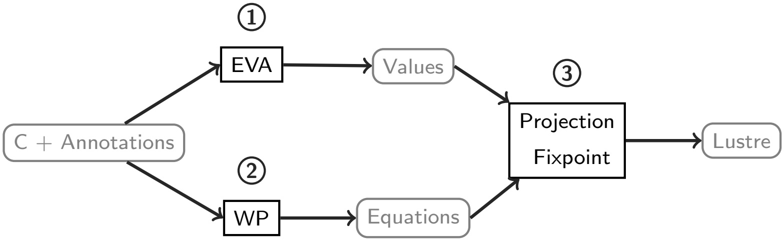


Figure: Extraction Methodology

Assumptions

- We expect a single top-level step function (and optionally an init function).
- We expect that all the loops in the step function are bounded.

Modular Extraction : example

Let's take a simple example:

```
int x, y, reset, state;

void sum(int *p, int x)
{
  *p += x;
}

/*@ input x, reset;
    output y; */
void counter(void)
{
  if (reset)
    state = 0;
  else
    sum(&state, x);
  y = state;
}
```

Modular Extraction : example

```c
void sum(int *p, int x)
{
  *p += x;
}
```

$$M' = M[p \mapsto M[p] + x] \qquad (WP)$$

Modular Extraction : example

```
void sum(int *p, int x)
{
  *p += x;
}
```

$$M' = M[p \mapsto M[p] + x] \qquad (WP)$$

$$M' = M[\&state \mapsto M[\&state] + x] \qquad (Eva)$$

Modular Extraction : example

```
void sum(int *p, int x)
{
  *p += x;
}
```

$$M' = M[p \mapsto M[p] + x] \qquad (WP)$$

$$M' = M[\&state \mapsto M[\&state] + x] \qquad (Eva)$$

$$i'_{state} = i_{state} + x \quad \textbf{with} \begin{cases} i_{state} = M[\&state] \\ i'_{state} = M'[\&state] \end{cases} \qquad (Projection)$$

Modular Extraction : Lustre generation

```c
int x, y, reset, state;

void sum(int *p, int x)
{
  *p += x;
}

/*@ input x, reset;
    output y; */
void counter(void)
{
  if (reset)
    state = 0;
  else
    sum(&state, x);
  y = state;
}
```

```
node sum_1(i_1, x_0: int) returns (i_2 : int)
let
  i_2 = i_1 + x_0;
tel

node counter(x, reset : int) returns (y : int)
let
  (* locals *)
  var i_0 : int;
  (* states *)
  i_1 = 0 -> pre(i_0);
  (* body *)
  if reset <> 0 then
    i_0 = 0;
  else
    i_0 = sum_1(i_1, x);
  (* outputs *)
  y = i_0;
tel
```

3. Verification Strategy

Verification Strategy: Lustre contracts with LustreSpec

```
node main(x:int) returns (y, z:int)
behavior B {
    assumes x_pos { x >= 0 }
    ensures y_pos { y >= x }
    ensures y_inc { y >= (0 -> pre(y)) }
}
begin
  if (x > 0) then {
    y = x;
    z = x;
  } else {
    y = 0 -> pre(y);
    z = -x;
  }
  check z_pos { z >= 0 }
end
```

Highly inspired by ACSL, Very similar to CoCoSpec [3]

Verification Strategy: Incremental Approach

- Properties expressed as observer flows
- Proof Engineering [4] approach for helping provers
- Incremental steps instead of single untractable proof
  - First prove property P_0,
  - then prove P_1 assuming P_0,
  - then prove P_2 assuming $P_0 \wedge P_1$

    ...

Verification Strategy: Incremental Approach

- Properties expressed as observer flows
- Proof Engineering [4] approach for helping provers
- Incremental steps instead of single untractable proof
  - First prove property P_0,
  - then prove P_1 assuming P_0,
  - then prove P_2 assuming $P_0 \wedge P_1$

    ...
- Some provers do not support this out of the box (GATeL)
- We encode this strategy as part of the export from LustreSpec to Lustre
  – (Annex section)

GATeL - Constraint Based Verification

- Generate test data given a model and an objective
  - Backward propagation of reachability objective at final cycle
  - Lustre/Scade flows: input/output clocked values over time
  - Reals, floats, modulo, delta, interval unions, clocks, etc.
  - Bounded by max number of cycles and numerical bounds
  - Detect and prove some patterns of K-induction

GATeL - Constraint Based Verification

- Generate test data given a model and an objective
  - Backward propagation of reachability objective at final cycle
  - Lustre/Scade flows: input/output clocked values over time
  - Reals, floats, modulo, delta, interval unions, clocks, etc.
  - Bounded by max number of cycles and numerical bounds
  - Detect and prove some patterns of K-induction
- Model compilation and simulation
  - Static simplifications assuming asserts
  - Statically detect linear growth and infer bounds wrt. cycle
  - Forward evaluator, over-approximating evaluator

GATeL - Constraint Based Verification

- Generate test data given a model and an objective
  - Backward propagation of reachability objective at final cycle
  - Lustre/Scade flows: input/output clocked values over time
  - Reals, floats, modulo, delta, interval unions, clocks, etc.
  - Bounded by max number of cycles and numerical bounds
  - Detect and prove some patterns of K-induction
- Model compilation and simulation
  - Static simplifications assuming asserts
  - Statically detect linear growth and infer bounds wrt. cycle
  - Forward evaluator, over-approximating evaluator
- COLIBRI - SMT solver
  - Colibri2 reimplementation in OCaml

4. Conclusion

Frama-C Synchrone

- Frama-C plugin started in 2015 (today roughly 10K of code)
  - Relies on Frama-C/EVA and Frama-C/WP for C extraction
  - Second implementation that better scales

Frama-C Synchrone

- Frama-C plugin started in 2015 (today roughly 10K of code)
  - Relies on FRAMA-C/EVA and FRAMA-C/WP for C extraction
  - Second implementation that better scales
- Transformations
  - C Code extractor
  - Lustre module Linker
  - Contract inlining: express contracts as `check` assertions
  - Proof observers: encode assertions for incremental proofs
  - Export to Lustre: use clocks (for now)

Frama-C Synchrone

- Frama-C plugin started in 2015 (today roughly 10K of code)
  - Relies on Frama-C/EVA and Frama-C/WP for C extraction
  - Second implementation that better scales
- Transformations
  - C Code extractor
  - Lustre module Linker
  - Contract inlining: express contracts as `check` assertions
  - Proof observers: encode assertions for incremental proofs
  - Export to Lustre: use clocks (for now)
- Case study from ASNR
  - Extraction from C code (approx. 50 functions)
  - Linked with specification
  - Export and proof with GATeL/Kind2

Related Work

- In [2], CoCoSim provides an automated framework to translate Simulink models to Lustre, for automated verification (Kind2 . . .)
  - Use a subset of Simulink close to Lustre.

Related Work

- In [2], CoCoSim provides an automated framework to translate Simulink models to Lustre, for automated verification (Kind2 . . .)
  - Use a subset of Simulink close to Lustre.
- In [6], CoCompiler provides a DSL lifter for Lustre from C code based on a rewriting of Velus using relational programming.
  - C program should however be really close to the compiler's image to be extracted.

Related Work

- In [2], CoCoSim provides an automated framework to translate Simulink models to Lustre, for automated verification (Kind2 . . .)
  - Use a subset of Simulink close to Lustre.
- In [6], CoCompiler provides a DSL lifter for Lustre from C code based on a rewriting of Velus using relational programming.
  - C program should however be really close to the compiler's image to be extracted.
- In [5], PsyC (a synchronous variation of C) are translated to Lustre, for automated verification (Kind2 . . .)
  - Only the synchronous primitives and the control-flow is translated to Lustre.

Conclusion and Future Work

FRAMA-C/SYNCHRONE:

- provides modular extraction of LUSTRE from synchronous-reactive C code.
- has a dedicated specification language inspired by ACSL function contracts, called LUSTRESPEC.
- has an incremental verification strategy inspired by FRAMA-C/WP using external model-checker tools: GATEL and KIND2

Conclusion and Future Work

Future Work:

- we plan to improve the modularity aspects of the verification process by reusing sub-nodes contracts as for compositional verification
- we plan to diversify the verification techniques:
  - by extracting complex invariants resulting from abstract interpretation
  - by implementing a dedicated WHY3 theory for proofs that are close to classical deductive verification.

References I

[1] B Blanc et al. 'Proving Properties of Reactive Programs From C to Lustre'. In: *ERTS 2018*. 2018.

[2] Hamza Bourbouh et al. 'CoCoSim, a code generation framework for control/command applications: An overview of CoCoSim for multi-periodic discrete Simulink models'. In: *Embedded Real Time Systems (ERTS) 2020* ARC-E-DAA-TN74591 (2020).

[3] Adrien Champion et al. 'CoCoSpec: a mode-aware contract language for reactive systems'. In: *International Conference on Software Engineering and Formal Methods*. Springer. 2016, pp. 347–366.

[4] Talia Ringer et al. 'QED at Large: A Survey of Engineering of Formally Verified Software'. In: *CoRR* abs/2003.06458 (2020). arXiv: 2003.06458. URL: https://arxiv.org/abs/2003.06458.

References II

[5] Fabien Siron. 'Methodology for the formal verification of temporal properties for real-time safety-critical applications based on logical time'. PhD thesis. Université Côte d'Azur, 2023.

[6] Naomi Spargo et al. 'The CoCompiler: DSL Lifting via Relational Compilation'. In: *arXiv preprint arXiv:2510.00210* (2025).

Verification Strategy: Encoding (1/2)

```
node main(x:int)
returns (y:int)
behavior B {
assumes x_pos {x >= 0}
ensures y_pos {y >= x}
ensures y_inc {y >= (0 -> pre(y))}
}
let
y = if (x > 0)
    then x
    else (0 -> pre(y));
tel
```

```
node __assume__ (goal: int;  bi: bool;  i: int) returns (p: bool)
let p = (not((i < goal)) or bi); tel

node __witness__ (goal: int;  bi: bool;  i: int) returns (p: bool)
let p = ((i = goal) and not(bi)); tel

node main (x: int;  __goal: int) returns (y: int;  __assumed: bool;  __reached: bool)
var
  __check_behavior_B_ensures_E1: bool;
  __check_behavior_B_ensures_E0: bool;
  __behavior_B: bool;
let
  __behavior_B = (x >= 0);
  y = if (x > 0) then x else (0 -> (pre y));
  __check_behavior_B_ensures_E0 = (not(__behavior_B) or (y >= x));
  __check_behavior_B_ensures_E1 = (not(__behavior_B) or (y >= (0 -> (pre y))));
  __assumed = (__assume__(__goal, __check_behavior_B_ensures_E1, 2) and
  __assume__(__goal, __check_behavior_B_ensures_E0, 1));
  __reached = (__witness__(__goal, __check_behavior_B_ensures_E1, 2) or
  __witness__(__goal, __check_behavior_B_ensures_E0, 1));
tel
```

Verification Strategy: Encoding (2/2)

```
node prove_main_1 (x: int) returns (y: int;  __property: bool)
var
  __witness: bool;
  __ind: bool;
  y0: int;
  __ind_1: bool;
  __witness_2: bool;
let
  y0, __ind_1, __witness_2 = main(x, 1);
  y = y0;
  __ind = __ind_1;
  __witness = __witness_2;
  assert __ind;
  __property = not(__witness);
tel
```

```
node prove_main_2 (x: int) returns (y: int;  __property: bool)
var
  __witness: bool;
  __ind: bool;
  y3: int;
  __ind_4: bool;
  __witness_5: bool;
let
  y3, __ind_4, __witness_5 = main(x, 2);
  y = y3;
  __ind = __ind_4;
  __witness = __witness_5;
  assert __ind;
  __property = not(__witness);
tel
```

GATeL - successful proof

```
node prove_main_1 (x: int) returns (y: int;  __property: bool)
var
  __witness: bool;
  __ind: bool;
  y0: int;
  __ind_1: bool;
  __witness_2: bool;
let
  y0, __ind_1, __witness_2 = main(x, 1);
  y = y0;
  __ind = __ind_1;
  __witness = __witness_2;
  assert __ind;
  __property = not(__witness);
tel

(* File "<unnamed buffer>", lines 2-8: *)
node prove_main_2 (x: int) returns (y: int;  __propert
var
  __witness: bool;
  __ind: bool;
  y3: int;
  __ind_4: bool;
  __witness_5: bool;
let
  y3, __ind_4, __witness_5 = main(x, 2);
  y = y3;
  __ind = __ind_4;
  __witness = __witness_5;
  assert __ind;
```

Proof Log

```
 and y<x)

*****************************************************************
* DECOMPOSITION INTO A DISJUNCTIION OF SUBGOALS (each subgoal mu


** DNF 1:
(x>=0
 and x>y)



*****************************************************************
* PROOF SESSION

Property leads to 1 DNF cases

Trying to prove DNF case 1/1
Proved (to be false) DNF case 1 (0.0 s)

Property has been proved to be valid (0.0399999999999991 s)
Postcondition of property is sat
```

GATeL - counterexample

```
  __check_behavior_B_ensures_E1 = (not(__behavior_B) or (y >= (0 -> (pre y))));
  __assumed = (__assume__(__goal, __check_behavior_B_ensures_E1, 2) and
  __assume__(__goal, __check_behavior_B_ensures_E0, 1));
  __reached = (__witness__(__goal, __check_behavior_B_ensures_E1, 2) or
  __witness__(__goal, __check_behavior_B_ensures_E0, 1));
tel

(* File "<unnamed buffer>", lines 2-8: *)
node prove_main_1 (x: int) returns (y: int; __property: bool)
var
  __witness: bool;
  __ind: bool;
  y0: int;
  __ind_1: bool;
  __witness_2: bool;
let
  y0, __ind_1, __witness_2 = main(x, 1);
  y = y0;
  __ind = __ind_1;
  __witness = __witness_2;
  assert __ind;
  __property = not(__witness);
tel

(* File "<unnamed buffer>", lines 2-8: *)
node prove_main_2 (x: int) returns (y: int; __property: bool)
var
  __witness: bool;
  __ind: bool;
  y3: int;
```

	1	0
x	1508	512
y	1508	512
__pr	true	false

Counter example found for DNF case 1 (0.0 s)

not(__property)