Efficient Generation of C Code from CCSL

Baptiste Allorant

ENS de Lyon, INRIA Sophia-Antipolis
under the supervision of Frédéric Mallet and Sid Touati

25 November 2021

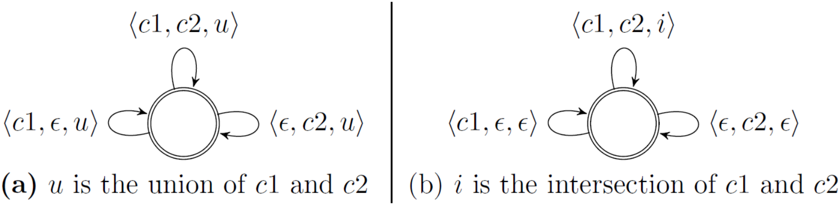
Presentation of CCSL

```
Specification Example{
    Clock a b c d e
    [
        Let A be a or b
        Let B be A and c
        Let C be B minus d
        Let D be inf(A, B)
        Let E be sup(A, B)

        repeat F every 4 C
        SubClocking e <- D
        Precedence C < E
        Precedence D <= E
        Exclusion d # e
    ]
}
```

Figure 1: LightCCSL Example

Semantic of CCSL



(a) u is the union of $c1$ and $c2$ | (b) i is the intersection of $c1$ and $c2$

Figure 2: Union and Intersection of Clocks in CCSL

Solving Strategies

- Compute the state automaton of the specification.

Solving Strategies

- Compute the state automaton of the specification.
- Use a SAT-Solver at each tick to compute the next tick.

Solving Strategies

- Compute the state automaton of the specification.
- Use a SAT-Solver at each tick to compute the next tick.

Is it possible to do better?

Solving Strategies

- Compute the state automaton of the specification.
- Use a SAT-Solver at each tick to compute the next tick.

Is it possible to do better?

The goal is to generate C code from a CCSL specification that will efficiently compute one solution to the specification.

Looking for possible Precomputations

Looking for possible Precomputations

Let C and D be a determined components.

$$
\begin{aligned}
Repeat\ a\ Every\ k\ b &: (a \in C \wedge b \in D) \rightarrow C = D \\
a = Def(b, c) &: (b \in D \wedge c \in D) \rightarrow a \in D
\end{aligned}
$$

Figure 3: Clocks in Determined Components.

Stateless Constraints

Stateless Constraints

- Not, Equivalence.

Stateless Constraints

- Not, Equivalence.
- Implication.

Stateless Constraints

- Not, Equivalence.
- Implication.
- Exclusion.

Stateless Constraints

- Not, Equivalence.
- Implication.
- Exclusion.
- Union, Intersection, Minus.

Stateless Constraints

- Not, Equivalence.
  - Union-Find.
- Implication.
- Exclusion.
- Union, Intersection, Minus.

Stateless Constraints

- Not, Equivalence.
  - Union-Find.
- Implication.
  - Implication Graph.
- Exclusion.

- Union, Intersection, Minus.

Stateless Constraints

- Not, Equivalence.
  - Union-Find.
- Implication.
  - Implication Graph.
- Exclusion.
  - Separation of A and $\neg A$.
- Union, Intersection, Minus.

Stateless Constraints

- Not, Equivalence.
  - Union-Find.
- Implication.
  - Implication Graph.
- Exclusion.
  - Separation of A and $\neg A$.
- Union, Intersection, Minus.
  - No simple Implication translation.

Union Case

$a = b \text{ or } c \Rightarrow$

$b \rightarrow a$

$c \rightarrow a$

Union Case

$a = b$ or $c \Rightarrow$

$b \rightarrow a$

$c \rightarrow a$

$a \rightarrow \neg b \rightarrow c$

$\neg b \rightarrow a \rightarrow c$

$\neg c \rightarrow a \rightarrow b$

Main Issue

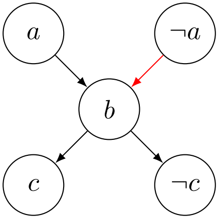


Figure 4: Main Issue

Issue from Union

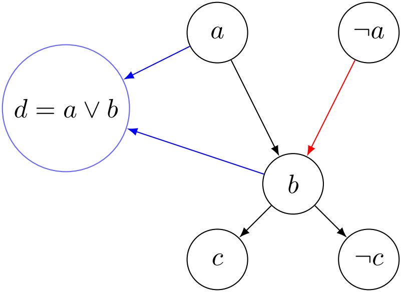


Figure 5: Union Issue

Issue from Minus

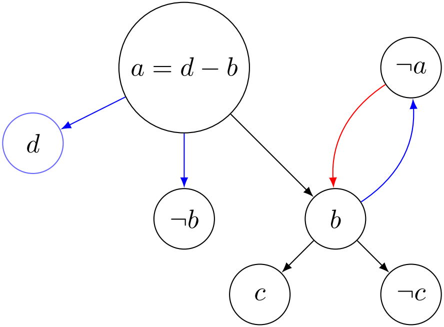


Figure 6: Minus Issue

The Order Approach

The Order Approach

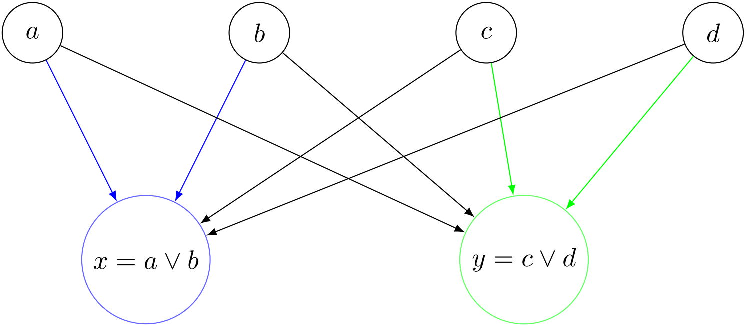


Figure 7: Order Counter-Example

The Backtrack Approach

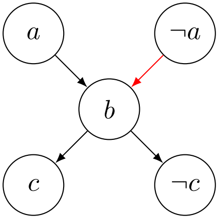


Figure 8: Basic Issue

The Backtrack Approach

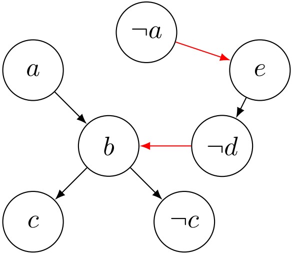


Figure 9: Backtrack Issue

Proof of NP-Completeness

Definition (Single-Step CCSL)

Let S be a CCSL specification, $n \in \mathbb{N}$, and T a valid schedule of S for the n first steps. Single-Step CCSL is defined as the problem of computing a valid $n + 1^{th}$ step.

Proof of NP-Completeness

Definition (Single-Step CCSL)

Let S be a CCSL specification, $n \in \mathbb{N}$, and T a valid schedule of S for the n first steps. Single-Step CCSL is defined as the problem of computing a valid $n + 1^{th}$ step.

Theorem

Single-Step CCSL is NP-Complete.

Proof of NP-Completeness

Definition (Single-Step CCSL)

Let S be a CCSL specification, $n \in \mathbb{N}$, and T a valid schedule of S for the n first steps. Single-Step CCSL is defined as the problem of computing a valid $n + 1^{th}$ step.

Theorem

Single-Step CCSL is NP-Complete.

Proof.

Not detailed here. □

What can I do now?

What can I do now?

1. Only one defined specification to solve.

What can I do now?

1. Only one defined specification to solve.
2. Simple counter-example can be avoided by simple static analysis.

What can I do now?

1. Only one defined specification to solve.
2. Simple counter-example can be avoided by simple static analysis.
3. Using many orders increases the size of the minimum counter-example.

What can I do now?

1. Only one defined specification to solve.
2. Simple counter-example can be avoided by simple static analysis.
3. Using many orders increases the size of the minimum counter-example.

The smallest counter-example just has to be bigger (for some measure) than the specification.

What can I do now?

1. Only one defined specification to solve.
2. Simple counter-example can be avoided by simple static analysis.
3. Using many orders increases the size of the minimum counter-example.

The smallest counter-example just has to be bigger (for some measure) than the specification.
Works in practice, still to be formalised...

Idea of the Architecture

1. Each alone constraint (or determined component) produces its current stateless constraint.
2. The solver gets the constraints and gives back an array with the result.
3. Each constraints check the array and updates its state if needed.

Optimisation of the Generated Code

Optimisation of the code using:

- Many compilers and compilation options.
- Two profilers: gprof and IntelVsyncProfiler.

Initial Version

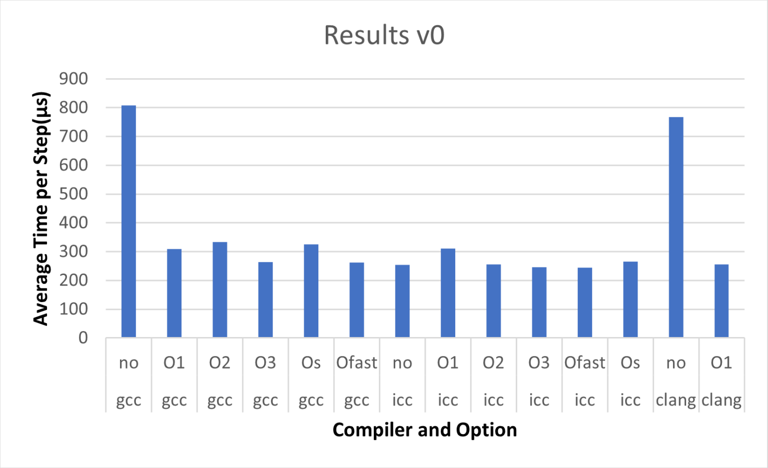


Figure 10: Performance of the Initial Code Generation

Final Version

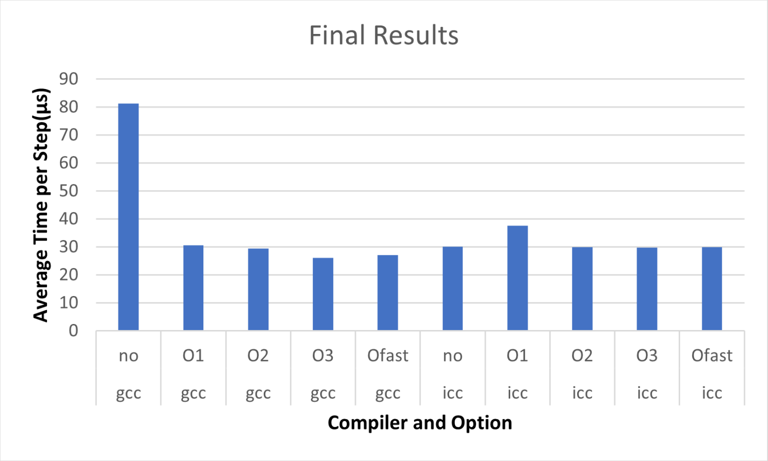


Figure 11: Performance of the Final Code Generation

Conclusion and Future Work

- Created an efficient and quite reliable code generator.

Conclusion and Future Work

- Created an efficient and quite reliable code generator.
- Some Performance Issues.

Conclusion and Future Work

- Created an efficient and quite reliable code generator.
- Some Performance Issues.
  - Need a wider evaluation campaign.

Conclusion and Future Work

- Created an efficient and quite reliable code generator.
- Some Performance Issues.
  - Need a wider evaluation campaign.
- Still correctness Issues.

Conclusion and Future Work

- Created an efficient and quite reliable code generator.
- Some Performance Issues.
  - Need a wider evaluation campaign.
- Still correctness Issues.
  - Formalise the minimum failure approach.

Conclusion and Future Work

- Created an efficient and quite reliable code generator.
- Some Performance Issues.
  - Need a wider evaluation campaign.
- Still correctness Issues.
  - Formalise the minimum failure approach.
  - Switch to LightC.

Conclusion and Future Work

- Created an efficient and quite reliable code generator.
- Some Performance Issues.
  - Need a wider evaluation campaign.
- Still correctness Issues.
  - Formalise the minimum failure approach.
  - Switch to LightC.
  - Deal with dead-ends.

Thank you very much for your attention.