Experience Report: Smuggling a Little Bit of Coq Inside a CAD Development Context

Dimitur Krustev

IGE+XAO Balkan

IGE+XAO GROUP

6 July 2020 / Coq Workshop 2020

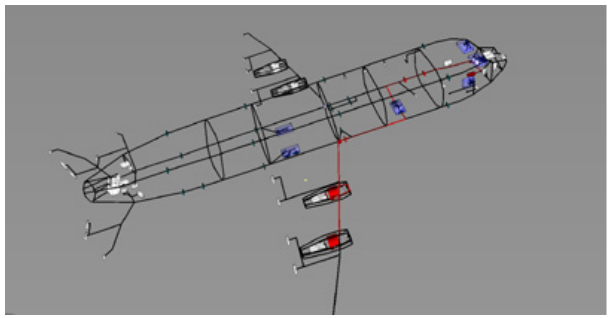
Outline

Introduction

- IGE+XAO – a company working on electrical CAD software for almost 35 years
  - a part of Schneider Electric since 2018
- Quality assurance based on a combination of widely used standard techniques
- However, we found formal verification using Coq useful in certain specific circumstances
  - why
  - when
  - how

Company – Products

- IGE+XAO – focus on electrical CAD systems, since 1986
- Solutions for several domains



  - Transport equipment manufacturing (Aircraft, Trains, Ships, Automotive)
    - system design of electrical installation
    - cable harness routing
    - cable harness manufacturing

      ...
  - Equipment, Machinery, Plant Automation
    - schematic editors
    - 3D Electrical Panel Design

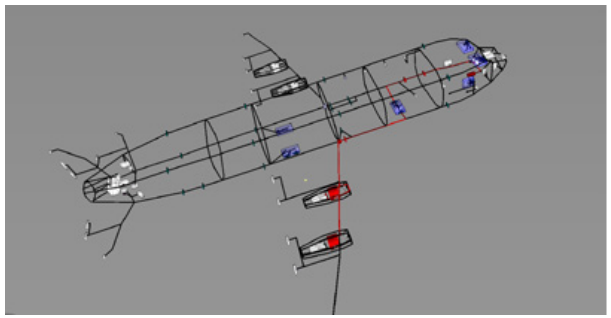
      ...
  - Construction

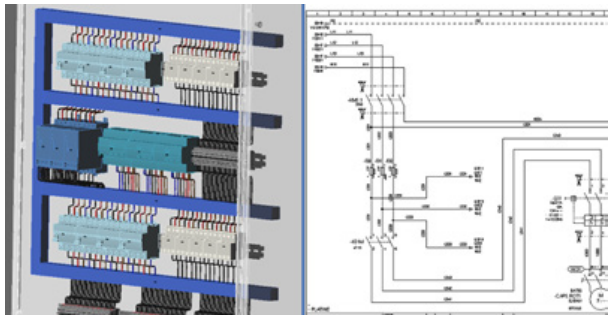
Company – Products

- IGE+XAO – focus on electrical CAD systems, since 1986
- Solutions for several domains
  - Transport equipment manufacturing (Aircraft, Trains, Ships, Automotive)
    - system design of electrical installation
    - cable harness routing
    - cable harness manufacturing

      . . .
  - Equipment, Machinery, Plant Automation
    - schematic editors
    - 3D Electrical Panel Design

      . . .
  - Construction





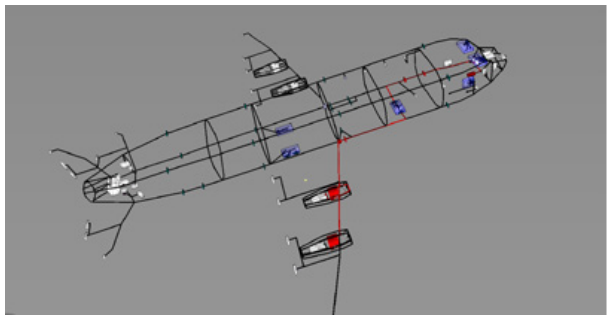
Company – Products

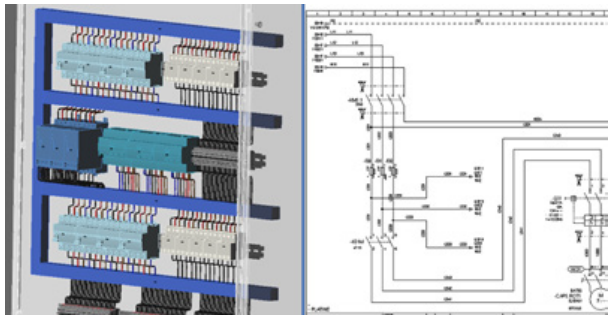
- IGE+XAO – focus on electrical CAD systems, since 1986
- Solutions for several domains

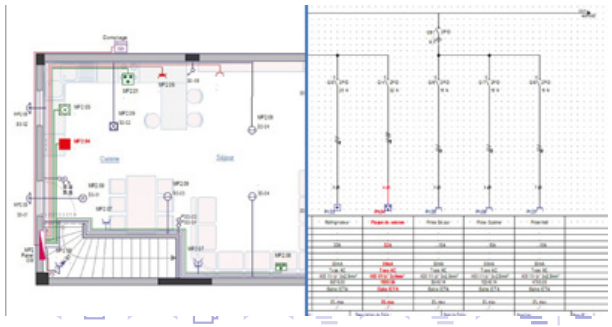
  - Transport equipment manufacturing (Aircraft, Trains, Ships, Automotive)
    - system design of electrical installation
    - cable harness routing
    - cable harness manufacturing

      . . .
  - Equipment, Machinery, Plant Automation
    - schematic editors
    - 3D Electrical Panel Design

      . . .
  - Construction





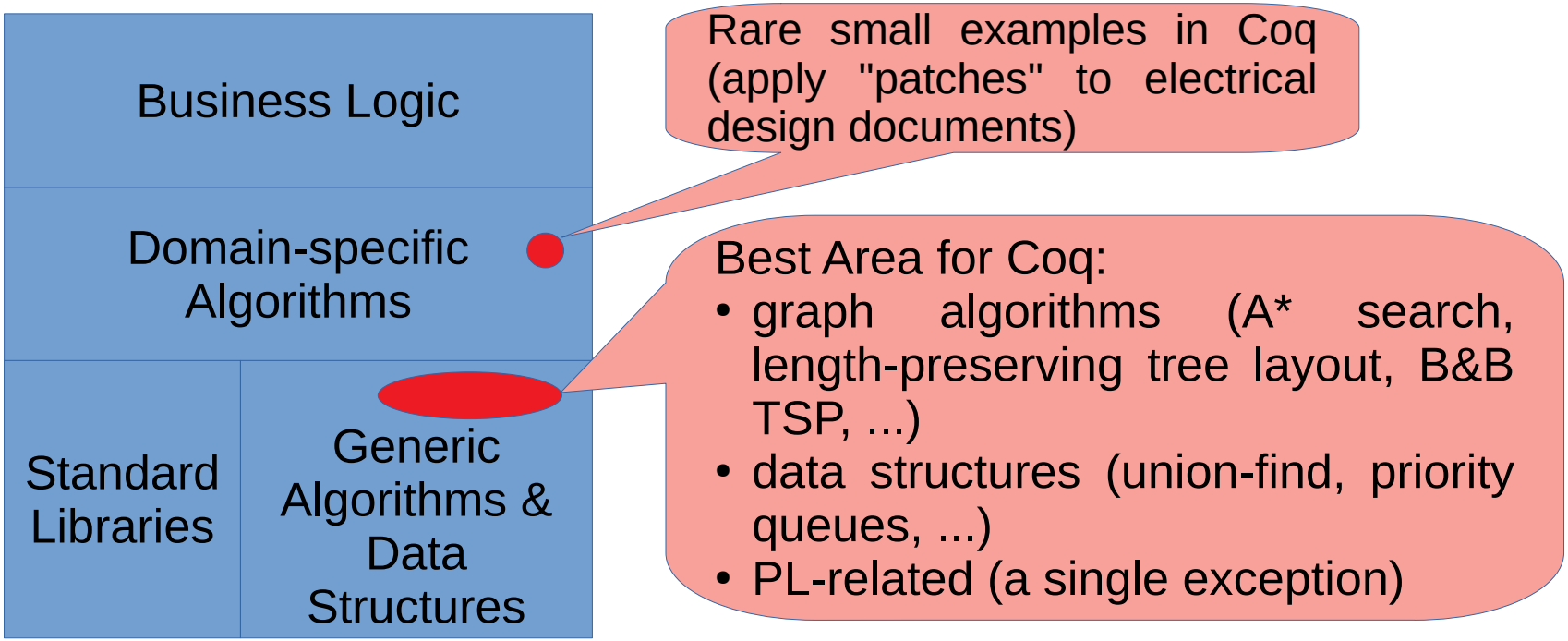


Company – Organization

- R&D departments in several countries
  - France, Poland, Bulgaria, Denmark, Tunisia
- Technologies used in recent years
  - majority of code still in C++
  - new projects based on .NET – mostly C#
  - more recently, F# also used in .NET projects
- QA – standard methods, expected to give best cost/quality ratio
  - unit/automated/manual tests
  - code reviews
  - code linters
- F# in our technology stack
  - faster to prototype domain-specific algorithms
  - immutable by default – easier to write correct parallel code
  - luckily, OCaml code extracted by Coq mostly usable as F# code

When We Use Coq

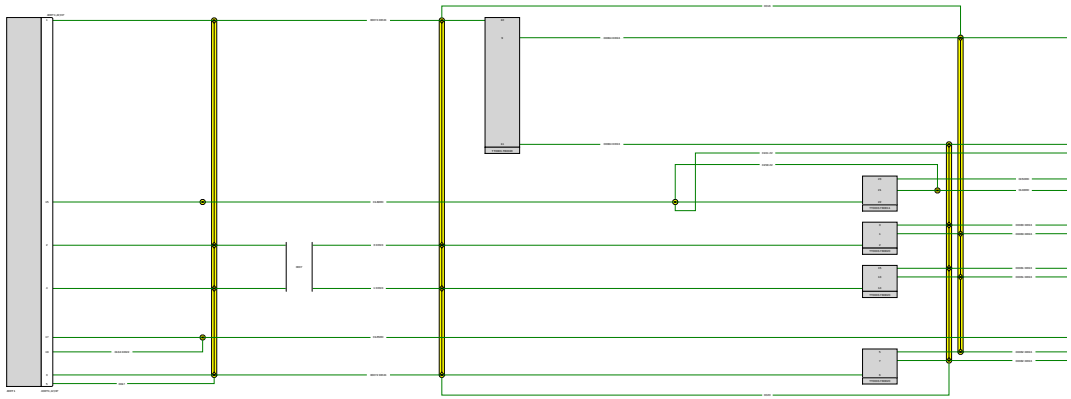
- Tricky generic algorithms – not in standard libraries – with disproportionately high impact on final quality
- Stable specification, easy to formalize



- Research work, not directly related to our production, is not discussed here

Example: A* Search – Context

- Context: a tool for automatically drawing wiring diagrams



- We needed a customized version of A* Search in order to find wire routes during diagram generation
  - to have more generic API (e.g. arbitrary edge weights)
  - to fine-tune performance (e.g. LIFO tie-breaking)

  $\Rightarrow$ an in-house implementation
- Subtle correctness arguments
- Key infrastructure for the whole product

$\Rightarrow$ We chose verification in Coq as main QA approach

Example: A* Search – Specification

```
Fixpoint CorrectRouteHelper (start: Node) (endNode: Node) (w:
    Weight) (path: list Node) : Prop :=
  match path with
  | nil => start = endNode ∧ w = weightZero
  | node::path => ∃ w', In (endNode, w') (neighbors node)
    ∧ ∃ w'', CorrectRouteHelper start node w'' path
    ∧ w = weightAdd w'' w' end.
Definition CorrectRoute (start: Node) (route: Node · Weight ·
    list Node) : Prop :=
  let '(endNode, w, path) := route in
  isGoalNode endNode = true ∧ NoDup (endNode::path)
  ∧ CorrectRouteHelper start endNode w path.
Theorem Astar_CorrectRoute: ∀ start route,
  Astar start = Some route → CorrectRoute start route.
```

- Relatively simple and not expected to change in the future
- Trade-off: only check result route correctness, not optimality

Example: A* Search – Evaluation

- Time spent on A* Search verification ($\sim$40h) only slightly longer that what would be needed to create initial implementation in F# with enough unit tests
- A* in Coq $\rightarrow$ A* in F# $\subset$ Wiring Diagram Generator (WDG) $\subset$ Electrical Diagram Visualizer (EDV)
- Top-level product extensively tested during 2 years:

	Language	Lines of code			Issues
		Impl.	Proofs	Cmts.	Issues
A*	Coq	173	203	29	**0**
A* (extracted)	F#	39	-	-	**0**
WDG	C# + F#	108K	-	-	400+
EDV	C# + TypeScript	-	-	-	800+

How We Use Coq

- Main goal: keep cost/quality ratio competitive with respect to other QA methods
- Avoid using tools/libraries not coming with the standard Coq installation
- Use built-in extraction to produce executable code
  - major enabler: we already use a language – F# – which is (mostly) compatible with Coq extraction
  - functional programming techniques already used in production – mostly because they make parallel programming easier
- Code verified in Coq typically tiny in size and stable over time

  $\Rightarrow$ so far, we can avoid Coq integration in automatic build process; integrating extracted code manually instead

Extraction: Technical Issues

- F# is compatible with OCaml core, but some features in extracted code are problematic
  - F# module system very limited ⇒ avoid using modules
  - no higher-kinded types ⇒ manual tweaking and/or some workarounds in Coq:

```
Record FinSetOps (A: Set) := { FinSet: Set; empty: FinSet;
 add: ∀ (A_dec: ∀ x y: A, {x = y} + {x <> y}), A → FinSet → FinSet;
 contains: ∀ (A_dec: ∀ x y: A, {x = y} + {x <> y}), A → FinSet → bool
   ; ... }.
Variable fsOps: FinSetOps Node.
```

higher-kinded type “hidden” in extracted code:

```
... let closedSet' = fsOps.add node_dec node closedSet ...
```

Why We Use Coq

- The use of Coq – for certain use cases – provides tangible net benefits in the long term[1]
  - short-term extra investment – need to spend time in doing proofs
  - short-term result – a 100% guarantee that the specification is respected (typically impossible with other QA methods)
  - long-term gains – no need to repeatedly deal with bugs, which inevitably appear regularly in tricky unverified code
    - typically far outweigh the short-term investment required
- Due to the nature of our products, use of formal verification can bring sufficient benefits only in a small number of situations, but the impact on quality is disproportionately high

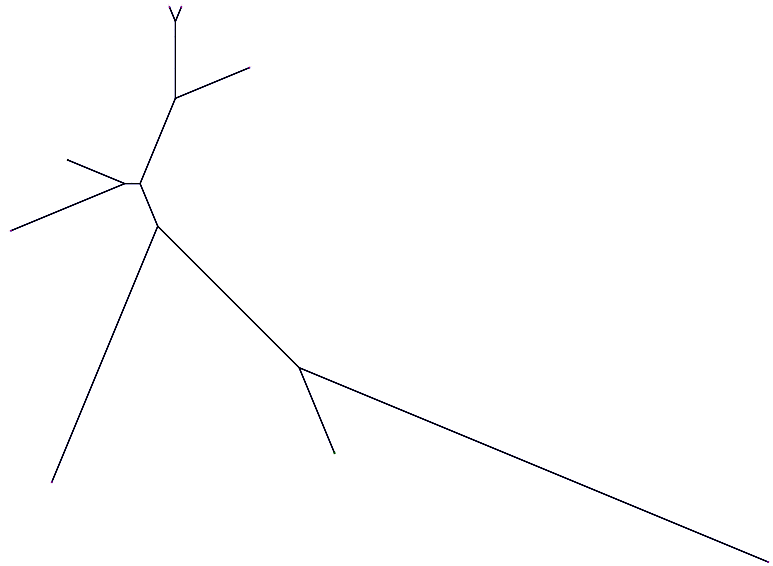
[1] assuming availability of competent Coq users

Conclusions

- Using Coq to formally verify selected parts of the code can be highly beneficial – in certain use cases – even for standard off-the-shelf software

Bonus Example: Tree Layout Preserving Lengths

- Early success (c. 2013)
- Context: Prepare cable harnesses for manufacturing on a table
- We found some existing algorithms designed for another domain (bioinformatics) – Bachmaier et al. 2005



- They needed adaptation for our domain

$\Rightarrow$ We successfully used Coq to verify our customized algorithm

```
Lemma layoutCountedTree_preservesLengths: ∀ ND ED getLen
    getCnt (t: Tree ND ED) a1 a2 x y, let t1 := layoutCountedTree
    getLen getCnt t a1 a2 x y in ∀ nd1 x1 y1 ed nd2 x2 y2, List.In
    ((nd1, (x1, y1)), ed, (nd2, (x2, y2))) (treeEdges t1) → (Rsqr (
    getLen ed) = Rsqr (x2 – x1) + Rsqr (y2 – y1))%R.
```