**Integrated Formal Approach for a Qualified Critical Code Generator**

**N. Izerrouken**\(^1\)\(^2\), M. Pantel, X. Thirioux and O. Ssi Yan Kai

\(^1\)Continental Automotive, Innovation Center, Toulouse, France
\(^2\)Institut de Recherche en Informatique de Toulouse, Institut National Polytechnique de Toulouse, France

---

**Problem Statement**

- Code generators reduce development time and verification costs (compliance of source code w.r.t. model-based design)
- Industry is aware that critical systems require more rigorous verification than classical testing
- Formal verification of specified properties (correctness of Block Sequencer, Typer, etc.)
- For each elementary tool:
  - Input model
  - Generated source code
- Compliance generated code/model
- Complex analysis of verification failures

**Context: GeneAuto**

- GeneAuto: Automatic code generator for critical embedded systems dedicated to transportation domain.
- GeneAuto is split into elementary tools
- Easier to specify, verify and validate
- Several implementations can be provided

**Main goals**

- Reduction of industrial unit testing costs
- Qualification of GeneAuto using DO178B/ED-12B recommendations
- Pragmatic integration of formal technologies into development tools for safety critical systems

**Integration of Formal Methods**

- Each elementary tool:
  - is developed and verified in Coq;
  - is verified and extracted in OCaml; extracted code preserves the properties proved in Coq.
- Java front-end:
  - reads input XML models;
  - executes extracted OCaml Wrapper;
  - writes output XML models.

- OCaml Wrapper:
  - reads input models
  - executes the extracted OCaml code (sequencer, typer, etc.);
  - writes the output result (execution order, types, etc.).

**Main Results**

- Mixing classical and formal development
- Development of correct-by-construction components
- ~4500 lines of Coq code and more than 130 proved theorems for the Block Sequencer
- Block Sequencer case study successfully integrated into GeneAuto
- Application to real-size systems from transportation domains

---

**Example of translation of requirements**

**From natural language**

- Properties of blocks based on data-flow restrictions.
- Sorted blocks with partial ordering according to priority from the input model.
- Sorted blocks that are still partially ordered according to their graphical position in the input model.

**To Coq language**

```
Definition correct_executions_order_dataflow
(m : Mode_Type) (s : SignalExecMode_Type) : Prop :=
  forall (d : nat), (d < d') /
  (forall i, i < |d| ->
    (forall d, d < d') /
    (forall d, d < d') ->
      (forall d, d < d') /
      (forall d, d < d')
```

**Qualification Concerns**

- Qualification process:
  - Qualification of the development process of Java components
  - Detailed documented development process using DO178B/ED-12B
  - Validation process done through testing and cross-reading
- Qualification of the formal elementary tools:
  - Coq proof checker partially verified
- Coq extractor generates OCaml code structurally similar to Coq specification
- Removal of unit & integration test phase from the formally developed elementary tools in DO178B/ED-12B

---

**Development Process**

**Choices**

- Specification, verification and validation using the Coq proof assistant
- Integration of the formal elementary tools to the GeneAuto tool chain
- Qualification of the development process of GeneAuto containing classical Java and formal elementary tools (Coq/OCaml)

**For each elementary tool**

- Translation of user/tool requirements from natural to formal language (complex task, human proof reading)
- Formal specification of the tool requirements and design
- Formal verification of specified properties (correctness of Block Sequencer, Typer, etc.)

**GeneAuto workflow**

- Definition of User Requirements
- Java Development
- Formal Methods Research
- Validation of User Needs
- Requirements
- Translated Requirements
- Modules
- Soft. quality requirements
- Results
- Qualification of GeneAuto