retour sur innovation

www.onera.fr
FORmal engineering of Critical Control-Command Embedded Systems

Virginie.Wiels@onera.fr, for the project team
With contributions from
Claire Pagetti and Thomas Loquen
Pierre-Loïc Garoche and Rémi Delmas
Overview

1. **Context and objectives**
2. Safe code generation
3. Formal verification
4. Integration
5. Conclusion
1. Context

- Certified embedded systems
  - Constraints on development / Verification objectives
  - Control-command: highly critical

- Challenges
  - Safety
  - And cost reduction

- Technical means
  - Automatisation
  - Early verification
  - Formal languages and techniques
1. Application target

- Targeted for civil aircrafts
  - With certification constraints

- Experimented on a UAV: « Avion jaune »
  - 11kg (empty)
  - 4m span
  - 4h autonomy
1. Objectives

- **Formal development chain**
  - Automated development
  - With verification at all stages
  - Dedicated to control-command systems
  - From control laws to embedded code

- **Global Optimisation**
  - Bridging the gap between system and software
  - Carrying information to ease formal verification all along the way

- **Incrementality**
  - Manage evolutions efficiently
Overview

1. Context and objectives
2. Safe code generation
3. Formal verification
4. Integration
5. Conclusion
2. Development process

Control design level

- Steps:
  - discrete models → choice of sampling periods
  - performance properties → definition of simulation tests
- Tools: Matlab / Simulink

Implementation level

- Steps:
  - coding of elementary blocks: Lustre
  - coding of multi-periodic assembly: Prelude
  - simulation with SchedMcore (simulation tests)

Integration level

- Steps:
  - Integration + test on the iron bird
  - Integration on the real UAV
2. From control design to implementation

- Elementary blocks → Lustre
  - gain, integrator, …

- Assembly → Prelude
  - Software architecture language
  - With real-time primitives

GENE-AUTO
- open-source code generation toolset (ITEA project)
  - input: Simulink, Stateflow or Scicos
  - output:
    - C or Ada;
    - Lustre (collaboration ONERA, IRIT, NASA)
  - limits: Matlab Simulink r2008b, discrete simulink, no fancy blocks, no complex z-expressions, only unit delays

ONERA toolbox
- Matlab library
  - input: Simulink
  - outputs: Lustre / Prelude
  - advantages: optimized for and specific to Onera controllers
  - drawback: development efforts
2. Lustre compiler (LustreC)

- Output: C or Java

- Open source implementation of Biernacki et al.


- Collaboration with IRIT-ENSEEIHT (X. Thirioux)
  Available at cavale.enseeiht.fr/redmine/projects/lustrec

- Compilation strategy
  - Same as Esterel Scade certified compiler
  - Less efficient than other compilation scheme but . . . used in industry and certifiable
  - Provide traceability (and verification means)
2. Simulation at implementation level

- **SchedMcore**
  open source ONERA tool
  [http://sites.onera.fr/schedmcore/](http://sites.onera.fr/schedmcore/)

- **Objective**
  - Simulation of the specification
  - Real-time constraints are assumed correct

- **Input**
  - Lustre and Prelude programs
    (several assemblies can be encoded)

- **Results**
  - Temporal simulations according to the scenario tests defined at control design level
  - if simulations are compliant,
    - high level properties are still satisfied
    - It simplifies the integration level

Longitudinal control
no significant degradation between
Simulink and SchedMcore results
Overview

1. Context and objectives
2. Safe code generation
3. Formal verification
4. Integration
5. Conclusion
3. Formal verification

• Which properties?
  • Stability verification
    • Open /closed loop
  • Safety architecture

• How?
  • Cooperation of formal verification techniques
  • Compiling specifications

• Collaboration with
  • NASA, Georgia Tech
  • Iowa University, Rockwell Collins
  • IRIT
3. Verification process

Control design level
- Simulink annotation blocks

Implementation level: model
- Taking into account the safety architecture
- Lustre contracts
- Model checking and abstract interpretation: TUFF, KIND

Integration level: source code
- ACSL contracts
- Frama-C
3. Safety architecture

Safety architecture

- Set of constructs to recover from system/hardware failures
- Redundancy
- Is this architecture sound? (i.e., when there is less than n simultaneous errors, the output is still valid or there will still be a working controller)
3. Verification: stability of control laws

- Open and closed loop stability
- At implementation level, taking into account redundancies
- Using information from analyzes at control design level

Lyapunov based stability
3. How? Existing verification techniques

- Mainly simulation and test
  - Not exhaustive
  - Costly (necessitates human expertise)

- Formal verification
  - Model checking: logical reasoning about abstraction of the system
    - SAT/SMT based model checking
    - Functional properties
  - Static analysis of the code
    - Mainly focused on numerical accuracy, data structure topology and manipulation

Limits exist for each of these techniques

=> Our proposal: cooperation of different techniques
Abstract Interpretation computes a sound bound (1.2) on each output whatever the value of the inputs.
Backward analysis applied on each triplex proves the specification
Assuming input is bounded by 1.2, we have output bounded by 3.6
Providing a bound on the inputs (3:6) an over-approximation of the output is computed: \( |u| \leq 194.499 \) using Lyapunov functions.
3. Example

\[
\begin{align*}
\left[-\infty, +\infty\right] & \quad 1.2 \quad 3.6 \quad 194.499
\end{align*}
\]
3. Cooperation of formal techniques

- Combining different formal verification techniques
  - To be able to analyze representative systems
    - Which cannot be analyzed by a single technique
  - Experimented on several case studies
- At model and at code level

TUFF – ONERA verification framework
3. Compiling specifications

node counter_simplespec (reset, active: bool) returns (safe: bool) ;
Let safe = reset => not active ; tel
--@ observer counter_simplespec (reset, active) ;
node counter reset : bool) returns (active: bool) ; ...

/*@ predicate counter_spec (bool reset, bool active) =
\let safe = reset => not active ;
safe ; */
//@ ensures counter_spec (reset, * active) ;
counter_step (_Bool reset , _Bool * active, counter_mem * self) { ... }
3. Synthesis

- Contracts (expressing properties) at each development level
- Cooperation of formal techniques for the verification of the contracts at a given level
- Compiling specifications
  - Traceability of requirements
  - Easier verification
Overview

1. Context and objectives
2. Safe code generation
3. Formal verification
4. Integration
5. Conclusion
4. Integration: two experimental platforms

- « Avion jaune »
  - UAV
  - With dedicated avionics

- Iron Bird
  - Same sensors/computers/actuators
  - System test bench
  - Allow early validation of development chain

Ongoing work on the development of a simulation framework for the iron bird
Overview

1. Context and objectives
2. Safe code generation
3. Formal verification
4. Integration
5. Conclusion
5. Conclusion

- **Initial objective**
  - Formal development and verification chain
  - Of control-command systems
  - For aircrafts
  - Experimented on UAV and associated iron bird

- **Could also be useful**
  - For development of control-command systems
  - For UAVs
    (especially in critical contexts requiring certification)