Overview of current concerns on the verification of numerical control systems

Pierre-Loïc Garoche – ENAC LII

Dec. 6th 2022

Au menu

FEANICSES project

Available methods and tools

Current Experiments: revalidating properties at code level

Directions for perspectives

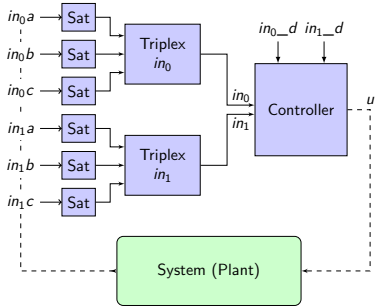
Au menu

FEANICSES project - initial goals

The FEANICSES project will expand the knowledge and practices of system-level property analysis at model and code level, providing a backbone for next-generation process development of critical CPS, with highly integrated formal methods.

Objective:
FEANICSES important challenge is the formal verification of system-level properties at all stages of a controlled system development.
Nowadays these properties are only evaluated at early stages when the controller is designed, and at the final stages, eg. when performing flight tests for an aircraft. Analyzing them formally and exhaustively all along the development chain will shorten the development process and leverage the quality of the final product, increasing the global safety of such systems.

Controllers



Typically:

- Linear core: $x_{k+1} = Ax_k + Bin_k$
- Non linearities:
  - Piecewise systems, ie. with modes:
    if $guard_i$ then $x_{k+1} = A_i x_k + B_i in_k$
  - Saturations on inputs/outputs
  - Linear Parameter Varying (LPV): linearization through gain interpolation
  - Polynomial update
  - Safety architecture (redundancy, voters, ...)

Hypothesis: everything is discrete, no continuous models (e.g. ODE)

Object under analysis – The Input

System:

- Code
  - set of functions, sequence of instructions, mix of boolean conditions, integer counters, floating point computations, pointers
  - no dynamic allocation (malloc), no nested loops
- Models
  - similar notions but simpler: no pointers, more types,
- knowledge can be provided on model components: a linear controller, an anti-windup, a saturation, etc

Safety property:

axiomatic semantics, aka predicate over values

- set of all reachable values is bounded
- a given bad region is unreachable
- high level properties: stability, robustness, bounded overshoot, etc
- ...

FEANICSES challenges

- C1: Express system level properties as numerical invariants
- C2: Address the analysis of complete controllers in their safety environment.
- C3: Provide new means to synthesize non linear invariants.
- C4: Adapt formal analysis methods to handle floating point soundness.
- C5: Develop a sound and scalable framework specific to the verification of controllers

Formal reasoning

- express the specification (in a formal way)

Formal reasoning

- express the specification (in a formal way)
  - what are the available/usable means to formalize properties of interest?

Formal reasoning

- express the specification (in a formal way)
  - what are the available/usable means to formalize properties of interest?
- use algorithms to address semantics analyses

Formal reasoning

- express the specification (in a formal way)
  - what are the available/usable means to formalize properties of interest?
- use algorithms to address semantics analyses.

Properties as boolean predicates p

$$p(x) \triangleq \begin{cases} x \in \mathbb{N} \setminus \{0,1\} \wedge \forall d \in \mathbb{N}, \\ x \ mod \ d = 0 \\ \quad \implies (d = x \vee d = 1) \end{cases}$$

Safety: $\forall s \in CS(S), p(s)$
Liveness: $\forall t \in TS(S), p(t)$

Properties as sets P

$x \in Primes = \{2, 3, 5, 7, 11, 13, \ldots\}$

Safety: $CS(S) \subseteq P$
Liveness: $TS(S) \subseteq P$

where

- $CS(S)$ denotes the collecting semantics of system S, i.e. it sets of reachable states
- $TS(S)$ its trace semantics, ie. its set of (possible infinite) traces

Approach 1 – Logical reasoning: $\forall s \in CS(S), p(s)$

- Express the transition system (Σ, I, T) as predicates: propositional encoding
  - Σ is a vector of variables x,
  - $I(x)$ is valid iff $x \in I$,
  - $T(x, y)$ is valid iff $(x, y) \in T$
- Express the property as a predicate $P(s)$
- Property as a satisfiability check
  - bounded horizon validity:
    prove that $\forall s_1, \ldots, s_k, I(s_0) \wedge \bigwedge_{0 \leq i < k} T(s_i, s_{i+1}) \implies (\bigwedge_{0 \leq j \leq k} P_j)\}$
  - infinite horizon, by induction[1]:
    - $\forall s, I(s) \implies P(s)$
    - $\forall s_1, s_2, T(s_1, s_2) \wedge P(s_1) \implies P(s_2)$
- Rely on external solvers (SMT) to verify the proof objectives

Can be used to verify imperative programs: Frama-C tool[2]

[1] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. "Checking Safety Properties Using Induction and a SAT-Solver". In: *FMCAD'00*. 2000, Aaron R. Bradley. "IC3 and beyond: Incremental, Inductive Verification". In: *CAV'12*. 2012.

[2] P. Cuoq et al. "Frama-C: a software analysis perspective". In: *SEFM'12*. Springer, 2012, pp. 233–247.

Approach 2 – Set-based reasoning: $CS(S) \subseteq P$

Collecting semantics as a fixpoint

Definition: Abstract Interpretation[3]

Abstract Interpretation is a constructive and sound theory for the approximation of semantics expressed as fixpoint of monotonic operators in a complete lattice.

[3] P. Cousot and R. Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *POPL'77*. 1977.

Approach 2 – Set-based reasoning: $CS(S) \subseteq P$

Collecting semantics as a fixpoint

Definition: Abstract Interpretation[3]

Abstract Interpretation is a constructive and sound theory for the approximation of semantics expressed as fixpoint of monotonic operators in a complete lattice.

(Collecting) Semantics ($\mathcal{R}$) as a fixpoint

- transition system: $(\Sigma, \mathit{Init}, \mathit{Step})$
- monotonic function:

$$\begin{array}{rcl} F : \wp(\Sigma) & \to & \wp(\Sigma) \\ X & \mapsto & \{s' \in \wp(\Sigma) | s' \in \mathit{Init} \vee \exists s \in X, (s, s') \in \mathit{Step}\} \end{array}$$

- $\mathcal{R} = \mathrm{lfp}\ F$

[3] P. Cousot and R. Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *POPL'77*. 1977.

Approach 2 – Set-based reasoning: $CS(S) \subseteq P$

Collecting semantics as a fixpoint

Definition: Abstract Interpretation[3]

Abstract Interpretation is a constructive and sound theory for the approximation of semantics expressed as fixpoint of monotonic operators in a complete lattice.

(Collecting) Semantics ($\mathcal{R}$) as a fixpoint

- transition system: $(\Sigma, \mathit{Init}, \mathit{Step})$
- monotonic function:

$$\begin{array}{rcl} F : \wp(\Sigma) & \to & \wp(\Sigma) \\ X & \mapsto & \{s' \in \wp(\Sigma) | s' \in \mathit{Init} \vee \exists s \in X, (s, s') \in \mathit{Step}\} \end{array}$$

- $\mathcal{R} = \mathrm{lfp}\ F$

Abstracting the collecting semantics: compute $\mathcal{R}^\#$ in an abstract domain $D^\# \subseteq \Sigma$

- $\alpha : \Sigma \mapsto D^\#$ and $\gamma : D^\# \mapsto \Sigma$
- $\mathcal{R} = \mathrm{lfp}\ F \quad \subseteq \quad \gamma \circ \mathcal{R}^\# \quad = \quad \gamma \circ \mathrm{lfp}\ F^\# \quad = \quad \gamma \circ \mathrm{lfp}\ (\gamma \circ F \circ \alpha)$

[3] P. Cousot and R. Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *POPL'77*. 1977.

More systems, more properties

More general systems

- Simple non linearities: Switches, Saturations, Antiwindups
- Non linear controllers, adaptive controllers
- Linear Parameter Varying (LPV), e.g. gain scheduled controllers
- Optimization-based control
- Neural Network based control
- Hybrid discrete-time continuous-time systems:
  - set-based simulation with paraboloids
  - co-simulation combining reachable tubes

More properties

- Formalize performance properties as invariants
  - e.g. bounded overshoot as H_∞ property
- Integral Quadratic Constraints to encode uncertainties and prove robustness

FEANICSES project - activities and results

PhDs:

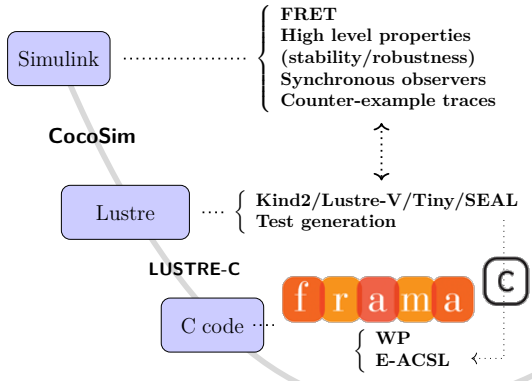
- Raphaël Cohen → verification of ellipsoid method QP (incl. floating point computation)
- Guillaume Davy → verification of an interior point algorithm (Primal LP)
- Hamza Bourbouh → CocoSim, Static analyses and model checking of mixed data-flow/control-flow models for critical systems
- Dylan Janak → Analysis and Synthesis of Safe Markov Decision Processes
- Sarah Li → Predicting and Regulating Learning Dynamics in Multi-agent Systems with Uncertainty (Markov Decision Processes)

Postdoc:

- Arash Sadeghzadeh → Reachable states computation for non linear systems
- Lelio Brun → generation of refinement proofs for Lustre [At ISAE with C. Garion and X. Thirioux]

Au menu

Integrated toolchain



- Proof of properties at model level
- Validation along the development cycle e.g. (re)proving Lyapunov functions at code level

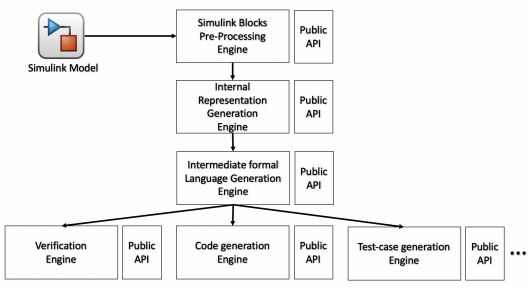
External tools

- Kind2 by U. of Iowa - `https://kind2-mc.github.io/kind2/`
  - model-checker and code generator
  - open-source, written in Ocaml
  - multiple model-checking algorithms and invariant synthesis methods
    - k-induction
    - PDR
    - blame and merit assignment, inductive validity cores, minimal cut sets, contract realizability checks, finite precision integers and more.
  - rely on external SMT solvers such as Z3, CVC4 to discharge the proof objectives
- Frama-C by CEA - `https://frama-c.com/`
  - Framework for the Modular Analysis of C
  - ACSL: Annotation language for C
    - very rich annotation language
    - axiomatic definitions, theorems
  - Multiple verification back-ends
    - WP: Weakest Precondition
    - E-ACSL: Executable ACSL

CocoSim - https://github.com/NASA-SW-VnV/CoCoSim

Temesghen Kahsai, Hamza Bourbouh, Andreas Katis

- formal semantics for a well-identified subset of Simulink/Stateflow
- fitted with a specification language (using synchronous observers) + FRET
- implemented as a compiler: phases
- main output: Lustre as an intermediate language
- integrated in Matlab Simulink: traceability, callbacks, signal builder, etc.
- supported by NASA
- open source



LustreC - https://github.com/Embedded-SW-VnV/lustrec

Xavier Thirioux, Christophe Garion, Lelio Brun

About 45kloc of OCaml - 2012–2022

- Language
  - Lustre v4
  - + custom flavored automatons and arrays
  - + annotation language CocoSpec[4]
- implements modular compilation scheme[5]
- designed to support verification along compilation
  - connected to verification tools
  - both at model level and code level

Open source - LGPL

[4] Adrien Champion et al. "CoCoSpec: A Mode-Aware Contract Language for Reactive Systems". In: *Software Engineering and Formal Methods, SEFM 2016*. 2016.

[5] D. Biernacki et al. "Clock-directed modular code generation for synchronous data-flow languages". In: *LCTES*. 2008, pp. 121–130.

LustreC - compiler

Compilation scheme

- automaton expansion as clocked expressions
- typing, clock computation
- normalization
- scheduling of expressions + computation of memories
- code generation

Produce

- C code
- + ACSL annotation (proof carrying code / refinement proof)

But also:

- external array types (no iterators, nor fancy lustre v4 extraction)
- machine types + MPFR (arbitrary precision floats)

LustreV - verifier

Rely on the compiler steps:

- multiple export targets
  - Simulink: support Simulink generation from Lustre
  - As switched polynomial system: amenable to Lyapunov function synthesis
  - As simple imperative code for static analysis: Tiny backend
  - As logical encoding: amenable to SMT based model-checking

```
1  [seal] DynSys: (2 memories, 3 init, 3 step switch cases)
2         Init: ...
3         Step:
4            [(((0.95 * __top_3) + (0.09975 * __top_2)) < -0.5)]:
5              __top_2 == ((-0.1 * __top_3) + (0.95 * __top_2));
6
7              __top_3 == -0.5;
8
9            [(((((0.95 * __top_3) + (0.09975 * __top_2)) >= -0.5) and (((0.95 * __top_3) +
                 (0.09975 * __top_2)) <= 0.5))]:
10             __top_2 == ((-0.1 * __top_3) + (0.95 * __top_2));
11
12             __top_3 == ((0.95 * __top_3) + (0.09975 * __top_2));
13
14           [(((0.95 * __top_3) + (0.09975 * __top_2)) > 0.5)]:
15             __top_2 == ((-0.1 * __top_3) + (0.95 * __top_2));
16
17             __top_3 == 0.5;
```

Tiny: a simple static analyzer

Very simple abstract interpreter

- input language is similar to While programming language
  - imperative code, while loops, no function calls
  - but external functions, eg. math functions
- Used for teaching
  - student write their own abstract domains and apply them on example
- Deal with integer, reals, booleans
  - affine and quadratic arithmetics, intervals, convex polyhedra (using Apron lib)

But, it provides useful features:

- bounds floating point errors (interval + error interval)
- partitioning abstract domains with arbitrary boolean expressions
- Fixed point engine with unrolling: supports bounded horizon set based simulation

Au menu

FEANICSES project

Available methods and tools

Current Experiments: revalidating properties at code level

Directions for perspectives

More properties: IQC

Current work with Virginia Tech: Prof. Mazen Farhood and Elias Khalife

Objective:

- express IQC analaysis, using a quadratic invariants, at code level
- including formal guarantees for floating point computation

Frama-C / WP / Alt-Ergo / OSDP

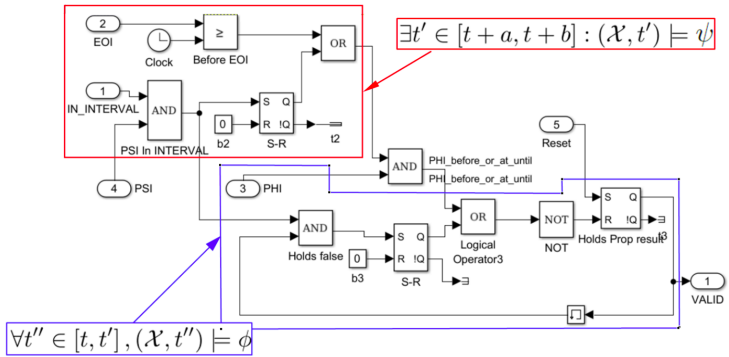
- IQC analysis and LMIs are currently computed in Matlab or Python
- code is manually produced and annotated
- tools are not yet fully integrated
  - Alt-Ergo/OSDP/Csdp
  - Tiny (for floating point error computation)

Au menu

Signal Temporal Logics as a verification tool

Numerous performance properties can be expressed as bounded horizon properties:

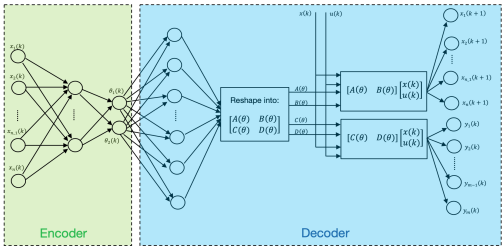
- STL as a Simulink library, using synchronous observers
- amenable to formal verification with model-checking
- linked with FRET?
- Eg. until block[6]



[6] A. Balsini et al. "Generation of simulink monitors for control applications from formal requirements". In: *2017 12th IEEE International Symposium on Industrial Embedded Systems (SIES)*. 2017.

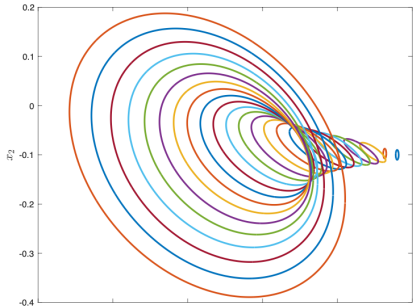
Sound embedding: non linear model reduction

- Different contexts require to abstract a function/program to support its analysis:
  - linearization: Linear Parameter Varying System (LPV)
  - Neural networks with non linear activation functions
  - internal computation involving a loop or a converging algorithm
- Abstract Interpretation and Deductive Method provide means to soundly manipulate programs
  - abstract the difficult part
  - work soundly on the abstracted program
  - Bound *err* the difference between f and its approximation $\tilde{f}$
  - Reasoning on $\tilde{f} + err$



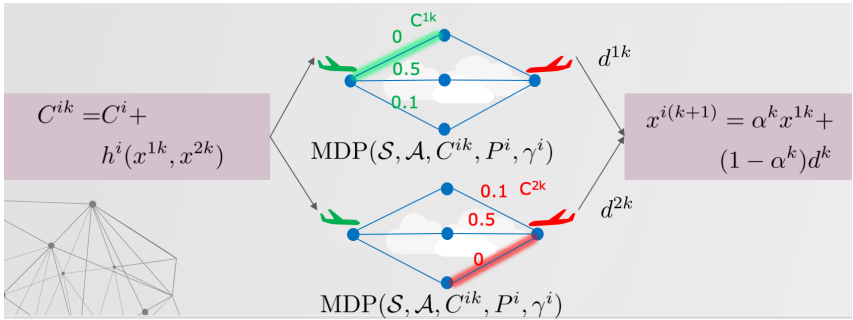
$$\begin{cases} \boldsymbol{x(k+1)} = \boldsymbol{f(x(k), u(k))} \\ \dots \end{cases}$$

$$\begin{cases} \boldsymbol{x(k+1)} = \boldsymbol{A(\theta(k))x(k) + B(\theta(k))u(k)} \\ \dots \end{cases}$$



Markov Decision Processes as a control tool

- Design MDP with guarantees
- application to modeling of Air Traffic Management
  - a framework to schedule paths while minmizing congestion
  - compatible with online rerouting, eg. due to unpredicted (weather) events
- One MDP for each flight. MDP congestion game for the air traffic planning
  - 74 flights originating from CDG and ORY between timestamps 39000 – 41000, July 3, 2017
  - 50 iterations of Frank-Wolfe to obtain the Nash equilibrium



Optimization-based control (MPC) - cf Pedro's talk

- Next generation of trajectory planning (Rocket landing, Rendez vous in space, drones, etc)
- Does not properly fit in the synchronous language paradigm (aka Lustre)
  - inner loops, with converging algorithms
  - but usually not explicit bounds on iterations

How to address the verification and valdiation of MPC

- Formal Specification?
- How to support code generation?

Optimization-based control (MPC) - cf Pedro's talk

- Next generation of trajectory planning (Rocket landing, Rendez vous in space, drones, etc)
- Does not properly fit in the synchronous language paradigm (aka Lustre)
  - inner loops, with converging algorithms
  - but usually not explicit bounds on iterations

How to address the verification and valdiation of MPC

- Formal Specification?
- How to support code generation?

More generally speaking: Do we need a dedicated language to design numerical algorithm?

- with verification means
- with autocoding means
- where basis type is vector, common functions are linear algebra operations, etc...
- like Python or Julia but with formal guarantees and code generation capabilities
- Can we rely on Lustre for part of the job?

End of FEANICSES workshop (and project)

Thank you all for attending!

Project ends in July 2023: a last workshop in may/june ?

How to continue interacting:

- share students
- send PhD students from one group to another
- develop collaborations within the group
  - Uli's automata + FRET + STL ?
  - IQC/Convex optimization based control/Reference Governors/LPV/...
- write new project proposal together?
  - around hybrid systems foundations and analysis?
  - code generation for embedded device?