Preserving Functional Correctness of Cyber-Physical System Controllers: From Model to Code

Guillaume Davy, Christophe Garion, Pierre-Loïc Garoche, Pierre Roux, Xavier Thirioux

Preserving functional correctness of cyber-physical system controllers: from model to code

Guillaume Davy*§, Christophe Garion†§, Pierre-Loïc Garoche*§, Pierre Roux*§ and Xavier Thirioux‡§
*Onera, The French Aerospace Lab, Toulouse, France †ISAE Supaero, Toulouse, France
‡IRIT INPT, Toulouse, France §University of Toulouse, France

***Abstract*—In this paper, we outline a methodology allowing to support the formal verification of functional properties for generated code. When relying on a code generator, a model is directly mapped into the target embedded code, in C for instance. At model level, a specification can be associated to the model and used to assess the validity of the model with respect to its requirements. At code level, other means such as deductive methods can be used to ensure similar goals. While the analysis of user-specified properties at model-level is developed and tractable, the automatic verification of these specification at code level remains an open issue. We present here a framework which builds a semantics layer connecting model specification to code specification, as well as associated proof evidences.**

This approach has been designed and developed in the context of dataflow languages such as Simulink, SCADE or Lustre, typically used in the design of cyber-physical system controllers, but it could also be revisited in other contexts. The model is analyzed by SMT-based model checking and convex optimization-based static analysis. At code level, deductive techniques, such as implemented in Frama-C, are used to prove the functional correctness. Our approach combines static analysis with refinement to drive the proof at code level, relying on analysis results obtained at model level. The refinement relates the initial model semantics with the one of the code.

This papers only outlines the methodology combining analyses. It has been applied manually on some examples. A fully implementation remains a future work.

***Index Terms*—Formal verification, Convex optimization, Cyberphysical systems, Compilers**

I. Context / Motivation

When developing critical systems, verification plays a major role and impacts drastically the process and the development cost. In recent years, formal methods have been received as an accepted means of compliance when performing verification and validation activities of critical software. This is implemented in certification norms such as DO-178C [16] for aerospace or EN50128 [13] for railway.

Another recent paradigm change is the widespread use of domain specific models to design embedded software. In some contexts, such as the design of control systems, models now enable a full characterization of the software which is eventually fully generated from the model design. This is for instance implemented through the MATLAB Simulink toolset which provides a model-based design environment fitted with autocoders producing embeddable code, or the ANSYS SCADE environment with similar features. In both cases, the model describes the composition of computation blocks, and relies on a dataflow semantics: an (un)specified sampling time is used to trigger the computation. Every time step the whole model is evaluated and this computation is assumed to be instantaneous. This model of computation is particularly suited for discrete time dynamical systems such as time-triggered controllers.

The use of models fitted with an executable semantics also enables early validation: the model can be simulated and evaluated with respect to its specification, before having the complete system.

Ideally, the requirements would have to be checked with respect to the final embedded code. Two different approaches can be used for this purpose. The first one trusts the compilation process to preserve the model semantics in the produced code and the verification activities can be performed only – or mainly – at model level. This is the approach currently used in the industry when relying on qualified compilers such as SCADE KCG [28] or developed in academia with proved compilers such as CompCert [23] or verified Lustre compilers [8], [22]. In these approaches, the compiler implementation is complex and relies on the formalization of a proof of the compilation process based on the Curry-Howard isomorphism.

A second approach, *translation validation* [29], [25], [26], authorizes to use of off-the-shelf compilers but requires the specification to be validated both at model and code level. This is the approach developed in this paper.

We propose to rely on formal methods and perform exhaustive verification of requirements at all design stages of the development process. Our approach is specification oriented and requires specification to be formally defined as a logical component. Formal analyses are then performed at model level to ensure the verification of the specification. The model is next compiled to embeddable C code and its specification checked again at code level.

The paper is structured as follows. Section II outlines the different approaches used to perform formal specification and verification. We also present basic algorithms used to support these verification activities. The next two sections focus on the model level: Section III addresses the formal specification of requirements at model level as well as the main principle behind model-checking, namely induction, while Section IV focuses specifically on the need to synthesize model invariants to support the verification activities. We present techniques to address the computation of non linear invariants using convex-based optimization. The last sections focus on code level:

Section V outlines the compilation process of both model and its specification components; Section VI presents our approach to automatize the proofs at code level using a refinement proof. Last, Section VII revisits the validation of numerical invariants at code level, accounting for numerical imprecision.

II. Preliminaries

We introduce in this section some definitions and outline the different tools and methods used to perform our formal analyses.

A. Notations: sets, logic and predicates

Sets are labeled with a capital letter (e.g. I, O, $\mathcal{V}$). For a set S, $\vec{S}$ (resp. $\mathbf{S}$) is the set of tuples (resp. streams) of elements of S. Elements in such sets are denoted as $\vec{s} \in \vec{S}$ and $\mathbf{s} \in \mathbf{S}$, where $\mathbf{s} = [s_0, s_1, \ldots] = \{s_i\}_{i \in \mathbb{N}}$, each s_i belonging to S.

Let $\mathcal{V}$ be a set of variables. Predicates, labeled in typewriter font (e.g. $\mathtt{p}$ or $\mathtt{init}$), are boolean functions over $\mathcal{V}$. We denote by $FV(\mathtt{p})$ the set of variables used in $\mathtt{p}$.

We denote by $\mathcal{M}^{\mathcal{V}} = (\mathcal{V} \to \mathbb{R})$ the possible models of a set of variables $\mathcal{V}$. A model associates a real value to each variable of $\mathcal{V}$. For a model $m \in \mathcal{M}^{\mathcal{V}}$, we interpret $\mathtt{p}[m]$ as the evaluation of predicate $\mathtt{p}$ with the model m, i.e. substituting all occurrences of free variables of $\mathtt{p}$ with their value in m.

B. Synchronous dataflow models

Synchronous languages are a class of languages proposed for the design of so called "reactive systems" – systems that maintain a permanent interaction with physical environment. Such languages are based on the theory of synchronous time, in which the system and its environment are considered to both view time with some "abstract" universal clock. In order to simplify reasoning about such systems, outputs are usually considered to be calculated instantly [5]. Examples of such languages include Esterel [6], Signal [2], the discrete subset of Simulink, or Lustre [12], [19]. In this paper, we focus on basic constructs and do not mention clock related features. In the following, we will use a syntax inspired from Lustre.

Programs manipulate streams, i.e. infinite sequences of values. At each time step, the system is considered to evaluate all streams, so all values are considered stable for any actual time spent in the instant between steps. A stream position can be used to indicate a specific value of a stream in a given instant, indexed by its clock tick. A stream at position 0 is in its initial configuration. Positions prior to this have no defined stream value. A dataflow program defines a set of equations of the form:

$$y_1, \ldots, y_n = f(x_1, \ldots, x_m, u_1, \ldots, u_o)$$

where y_i are output or local variables and u_i are input variables. Variables are used to represent individual streams and they are typed with basic types including streams of *Real* numbers, *Integers*, and *Booleans*. Programs and subprograms are expressed in terms of *Nodes*. Nodes directly model subsystems in a modular fashion, with an externally visible set of inputs and outputs. A node can be seen as a mapping of a finite set of input streams (in the form of a tuple) to a finite set of output streams (also expressed as a tuple). The *top node* is the main node of the program, the one that interfaces with the environment of the program and is never called by another node.

At each instant t, each node takes in the values of its input streams and returns the values of its output streams. Operationally, a node has a cyclic behavior: at each cycle t, it takes as input the value of each input stream at position or instant t, and returns the value of each output stream at instant t. Nodes have a limited form of memory: when computing the output values they can also look at input and output values from the previous instant. Nested definitions allow to access up to a finite limit of history, statically determined by the program itself.

The body of a node consists typically in a set of definitions, i.e. stream equations of the form $x = t$ where x is a variable denoting an output or a locally defined stream and t is an expression, in a certain stream algebra, in which variables name input, output, or local streams. More generally, x can be a tuple of stream variables and t an expression evaluating to a tuple of the same type. Most of dataflow operators are point-wise lifting to streams of the usual operators over stream values. For example, let $\mathbf{x} = [x_0, x_1, \ldots]$ and $\mathbf{y} = [y_0, y_1, \ldots]$ be two integer streams. Then, $\mathbf{x} + \mathbf{y}$ denotes the stream $[x_0 + y_0, x_1 + y_1, \ldots]$. An integer constant $\mathbf{c}$ may denote the constant integer stream $[c, c, \ldots]$. Two important additional operators are defined: a unary shift-right operator *pre* ("previous"), and a binary initialization operator $\to$ ("followed by"). *pre* is defined as $pre(\mathbf{x}) = [u, x_0, x_1, \ldots]$ with the value u left unspecified. $\to$ is defined as $\mathbf{x} \to \mathbf{y} = [x_0, y_1, y_2, \ldots]$. Syntactical restrictions on the equations in a program guarantee that all its streams are well defined: e.g. forbidding recursive definitions hence avoiding algebraic loops.

Definition 1 (Node definition)**.** *A Lustre node N is defined by the tuple (I_N, O_N, L_N, Eq_N) where $I_N, O_N, L_N \subseteq \mathcal{V}$ denote set of input, output and local variables, respectively, and Eq_N is a set of equations in $\mathcal{L}(I_N \cup O_N \cup L_N)$ defining the Lustre node streams where $\mathcal{L}$ denotes the set of valid expressions as defined previously.*

The *pre* construct characterizes local memories of a node. Since a node can call other nodes which can themselves contain memories, we denote a node state by the treedescribing this structured set of local memories. Each call to a node characterizes an instance of it, identified by a unique identifier (uid). We annotate the call by this uid.

Definition 2 (Node State S_N)**.** *Let $\mathcal{N}$ be a set of nodes and (I_N, O_N, L_N, Eq_N) be a node $N \in \mathcal{N}$. Let Uid a set of uid and $Uid*$ its Kleene closure with $\emptyset$ the empty word. We assume that each occurrence of a call in an equation $eq \in Eq_N$ has been associated to a uid $\in Uid$. We define recursively a node*

state as the set $S_N \in \wp(Uid* \times \mathcal{V})$

$$S_N \triangleq \begin{array}{c} \{(\emptyset, v) \mid v = pre\ e \in Eq_N\} \\ \cup \\ \left\{ (uid \cdot uid_v, v) \;\middle|\; \begin{array}{l} v = N'^{uid}(e) \in Eq_N \wedge \\ (uid_v, v) \in S_{N'} \end{array} \right\} \end{array}$$

To ease the following notations, we consider this set as a regular set of variables, $S_N \subseteq \mathcal{V}$*, provided an appropriate injective naming scheme.*

C. Axiomatic semantics

As introduced by Hoare [21], specification of imperative programs can be formally expressed using predicates via *axiomatic semantics*. When expressing requirements about an imperative program, a Hoare triple (`Pre`, Code, `Post`) can be used to define an assume/guarantee contract. Both `Pre` and `Post` are predicates over the Code variables and the contract should be understood as "When Pre is valid before the execution of Code then Post holds afterwards". When manipulating C code, ACSL, the ANSI C specification language [4], provides means to write such specifications and to attach them to code with annotations. These annotations are defined as code comments and do not modify the behavior of the final code. However, they can be manipulated by specific verification algorithms and used to check the validity of the specified contracts.

D. Tools

1) Satisfiability Modulo Theory (SMT): SAT solvers analyze propositional formulas and return a *SATisfiable assignement*, that is, a valuation of all propositional variables that render the propositional formula valid. When no such assignment is feasible the provided formula is identified as *unsatisfiable* (UNSAT). SMT solvers extend SAT solvers, enabling the atoms of the propositional formulas to be expressed in first-order theories, for instance linear real arithmetics.

2) Convex optimization: Convex optimization is a restriction of general numerical optimization problems. A convex optimization problem is defined as follows:

$$\begin{array}{ll} \min & f_0(x) \\ \text{s.t.} & f_i(x) \leq 0 \text{ for } i \in [1, m] \\ & a_j^\intercal x = b_j \text{ for } j \in [1, p] \end{array}$$

where f_0 and all f_i are convex functions and a_j, b_j vectors. A well known special case convex optimization is linear optimization or LP (linear programming) in which the objective function f_0 and the constraints f_i are linear.

This notion of convex optimization can be extended to more general convex sets. For instance, using semi-definite matrices (a matrix A is positive semidefinite, denoted by $A \succeq 0$, iff $\forall x, x^\intercal A\, x \geq 0$) leads to Linear Matrix Inequalities (LMI) for f_i constraints and SDP (semi-definite programming) solvers, in which unknown variables denote matrices, can be used to solve such problems.

Another equivalent kind of convex optimization problems is SOS (Sums-of-Squares) programming. Showing positivity of a polynomial constraint can be done by rewriting it as a – usually large – LMI problem. Positive constraints over polynomial can then be solved using SDP solvers.

III. Model analyses: Formal specification and verification of synchronous dataflow models

A. Propositional encoding

Relying on the definition of node equations and the characterization of node states, a node semantics can be encoded as a pair of predicates $\texttt{init}_\texttt{N}[s \in \overrightarrow{S_N}]$ and $\texttt{step}_\texttt{N}[i, o, s, s' \in \overrightarrow{I_N} \times \overrightarrow{O_N} \times \overrightarrow{S_N} \times \overrightarrow{S_N}]$ (cf. [17]).

Definition 3 (Node propositional encoding)**.** *Let* $\mathcal{N}$ *be a set of nodes and* (I_N, O_N, L_N, Eq_N) *be a node* $N \in \mathcal{N}$*. We define the semantics of* N *through the following two predicates:*

- $\texttt{init}_\texttt{N}[s \in \overrightarrow{S_N}]$*;*
- $\texttt{step}_\texttt{N}[i, o, s, s' \in \overrightarrow{I_N} \times \overrightarrow{O_N} \times \overrightarrow{S_N} \times \overrightarrow{S_N}]$

as the smallest predicates (wrt logical implication) verifying, for all input streams $\mathbf{i} \in \overrightarrow{\mathbf{I_N}}$ *and output streams* $\mathbf{o} \in \overrightarrow{\mathbf{O_N}}$*, the property:*

$$\exists \mathbf{s} \in \overrightarrow{\mathbf{S_N}}\ .s.t.\ \texttt{init}_\texttt{N}[s_0] \bigwedge_{i \in \mathbb{N}} \texttt{step}_\texttt{N}[i_i, o_i, s_i, s_{i+1}]$$

B. Synchronous observers

A *synchronous observer* [20], [38], [36] is a wrapper dataflow node used to test observable properties of a node N with minimal modification of the node itself. It returns an error signal if the property does not hold, reducing the more complicated property to a single Boolean stream. This stream has just to be checked to be constantly true. To support such specification during the compilation process, we have extended the traditional Lustre language with annotations similar to ACSL ones, Lustre contracts. Assume/guarantee formulas can be expressed in Lustre contracts as follows:

```
(* @ requires Pre(i);
     ensures  Post(i, o); *)
node N (i : I_N) returns (o : O_N);
```

where Pre and $Post$ are Lustre boolean expressions representing respectively the assumptions and guarantees of the contract.

Definition 4 (Contract satisfaction)**.** *We say that a node* N ***fulfills its contract*** *(denoted by* $N \models \langle Pre, Post \rangle$*) if and only if, for all input, output and state sequences* $\mathbf{i} \in \overrightarrow{\mathbf{I_N}}, \mathbf{o} \in \overrightarrow{\mathbf{O_N}}, \mathbf{s} \in \overrightarrow{\mathbf{S_N}}$ *the following holds:*

$$\forall j, (\texttt{init}_\texttt{N}[s_0] \wedge \forall k \in [0..j].\texttt{step}_\texttt{N}[i_k, o_k, s_k, s_{k+1}] \wedge \texttt{Pre}[i_k]) \implies \texttt{Post}[i_j, o_j]$$

Intuitively, the above definition asserts that if the assumptions have held at all instants up and including the current time, then the guarantee holds at the current time. Such formulation follows closely the contract semantics defined in the AGREE framework [14].

Remark 1 (Expression over node states)**.** *Without loss of generality, we can express these contracts over node states.*

In the following developments, to keep the presentation simpler, we deliberately expressed contracts $\langle \texttt{Pre}, \texttt{Post} \rangle$ over system states. Expressions $\texttt{Pre}[\mathbf{s}]$ and $\texttt{Post}[\mathbf{s}, \mathbf{s}']$ are then predicate over elements of S_N.

C. Formalizing specification.

Synchronous observers are a powerful way to express properties, as long as they can be expressed over state variables values. When analyzing control systems, two main issues are faced. First, most properties are expressed on the closed loop description of the system, i.e. they integrate the plant model used in the design. A solution would be to discretize the plant model and treat it as a regular program [34]. It could then be embedded within the observers, building complex stateful observers. A second and more difficult issue is the characterization of properties in the frequency domain [3]. This typically happens when analyzing linear control system and is largely used in the control community. A possible solution is to rely on the Kalman-Yakubovich-Popov [30], [31] lemma which relates frequency domain properties to linear matrix inequalities, i.e. numerical invariants over state variables. As an example, we have revisited the computation of phase and gain margin using this lemma and characterized the appropriate numerical property [37].

D. SMT-based model-checking: Contract verification with induction

Let us now assume that the semantics of a model node N is described by a predicate $\texttt{step}_\texttt{N}[\mathbf{i}, \mathbf{o}, \mathbf{s}, \mathbf{s}']$[1] defining the relationship between input flows $\mathbf{i}$, output flows $\mathbf{o}$ as well as internal states $\mathbf{s}$, as defined in Eq_N for a Lustre node. Let $\texttt{init}_\texttt{N}[\mathbf{s}]$ be the predicate over initial states.

As specification is described as a synchronous observer, similar predicates can be defined for the `Pre` and `Post` conditions since both of them are expressed as regular Lustre equations.

Proving exhaustively the validity of the contract (cf Def. 4) with respect to the node semantics can be performed by induction:

$$\texttt{init}_\texttt{N}[s] \wedge \texttt{step}_\texttt{N}[i, o, s, s'] \wedge \texttt{Pre}[s] \implies \texttt{Post}[s'] \quad (1)$$

$$\texttt{step}_\texttt{N}[i, o, s, s'] \wedge \texttt{Pre}[s] \wedge \texttt{Post}[s] \implies \texttt{Post}[\tilde{s'}] \quad (2)$$

Both properties are universally quantified over free variables s, s', i, o. As outlined in Sec. II-D1 they can be verified using SMT-based model-checking. SMT-based model-checking tools rely on more sophisticated variants of this principle, for example with algorithms such as k-induction [24] or PDR/IC3 [10]. In all cases the analysis expresses both the model semantics and the property as predicates and achieves an induction proof.

[1]We refer to the traditional definition of transition system in model checking techniques. A detailed description of a transition system for Lustre programs can be found in [18].

IV. Invariant synthesis based on Convex-based optimization, Lyapunov function and positive invariance

Unfortunately, most valid properties are not inductive with respect to the model semantics. Let us consider the collecting semantics of N, defined as the set of models $Coll_N \subseteq \mathcal{M}^{S_N}$:

$$\left\{ s_n \;\middle|\; \exists \begin{array}{l} i_0, \ldots, i_n \in \mathcal{M}^{I_N}, \\ o_0, \ldots, o_n \in \mathcal{M}^{O_N}, \\ s_0, \ldots, s_{n-1} \in \mathcal{M}^{S_N}, \end{array} \begin{array}{l} \texttt{init}_\texttt{N}[s_0] \wedge \\ \bigwedge_{0 \le i < n} \texttt{step}_\texttt{N}[i_i, o_i, s_i, s_{i+1}] \end{array} \right\}$$

We define the associated predicate $\texttt{coll}_\texttt{N}[s] = s \in Coll_N$.

Any valid property on the model is verified on all reachable states, in other words, it is inductive with respect to the collecting semantics. In that setting the induction principle is complete.

$$\begin{array}{l} \texttt{coll}_\texttt{N}[s] \wedge \texttt{coll}_\texttt{N}[s'] \wedge \texttt{init}_\texttt{N}[s] \\ \wedge\ \texttt{step}_\texttt{N}[i, o, s, s'] \wedge \texttt{Pre}[s] \end{array} \implies \texttt{Post}[s'] \quad (3)$$

$$\begin{array}{l} \texttt{coll}_\texttt{N}[s] \wedge \texttt{coll}_\texttt{N}[s'] \\ \wedge\ \texttt{step}_\texttt{N}[i, o, s, s'] \wedge \texttt{Pre}[s] \wedge \texttt{Post}[s] \end{array} \implies \texttt{Post}[\tilde{s'}] \quad (4)$$

The same contract $(\texttt{Pre}, \texttt{Post})$ may not be inductive over some arbitrary states s, s.t. $\neg\texttt{coll}_\texttt{N}[s]$. Such a state would correspond to a spurious counter-example: a state s_1 unreachable but satisfying `Post` such that its successor s_2 by the transition system semantics violates `Post`: $(\texttt{Pre}[s_1] \wedge \texttt{Post}[s_1] \wedge \texttt{step}_\texttt{N}[i_1, o_1, s_1, s_2]) \not\Rightarrow \texttt{Post}[s_2]$

While $Coll_N$ is, in general, not computable, any over-approximation of it, i.e. an invariant $\texttt{inv}[s]$ such that $\forall s, \texttt{coll}_\texttt{N}[s] \implies \texttt{inv}[s]$, could be used to reinforce the model description.

Substituting $\texttt{coll}_\texttt{N}[s]$ by a computable $\texttt{inv}[s]$ in Eqs (3) and (4) reinforces the induction proof but introduces incompleteness (since `true` is a trivial over-approximation of $\texttt{coll}_\texttt{N}[s]$).

A. Static analysis – Abstract interpretation

Static analysis, and more specifically abstract interpretation, provide means to compute over-approximation of the collecting semantics. The theory relies on Tarski fixpoint characterization and identifies the (non computable) collecting semantics as the set $Coll_N$ defined by a least fixpoint

$$Coll_N = lfp_\perp F = min_{X \subseteq \mathcal{M}^{S_N}} \{X | F(X) \subseteq X\} \quad (5)$$

of the following function F manipulating sets of models M:

$$\begin{array}{l} F : \wp(\mathcal{M}^{S_N}) \to \wp(\mathcal{M}^{S_N}) \\ \quad M \mapsto \left\{ m' \;\middle|\; \begin{array}{l} init_N[m'] \vee \\ \exists i \in \mathcal{M}^{I_N}, o \in \mathcal{M}^{O_N}, m \in M, \\ \text{s.t. } step_N[i, o, m, m'] \end{array} \right\} \end{array} \quad (6)$$

As a consequence, any set C of models $\mathcal{M}^{S_N}$ verifying the condition $F(C) \subseteq C$ is a sound over-approximation of $Coll_N$ since $Coll_N \subseteq C$ and therefore $\forall s \in \mathcal{M}^{S_N}, \texttt{coll}_\texttt{N}[s] \implies s \in C$.

Abstract interpretation attempts to compute such a set C by considering a family of such sets, notorious ones being intervals ($C \triangleq S_N \to \mathbb{R} \times \mathbb{R}$) or convex polyhedra ($C \triangleq (Ax +$

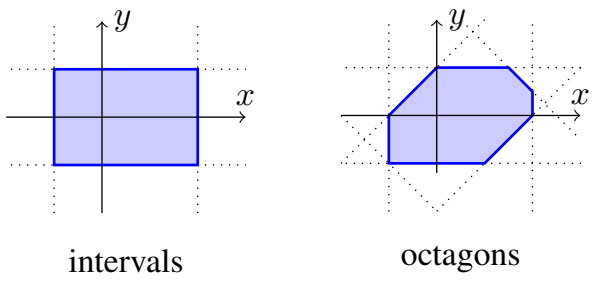


Fig. 1: Examples of abstract domains.

$b \leq 0$) with $x \in S_N, n = card(S_N), A \in \mathbb{R}^n \times \mathbb{R}^n, b \in \mathbb{R}^n$). Fig. 1 illustrates such computed sets, called abstract domains, here an interval-based set and an octagon.

B. Lyapunov functions and positive invariant sets

In 1890, Alexander LYAPUNOV published his well known result stating that the differential equation $\frac{dx}{dt} = Ax(t)$ is stable iff a positive-definite matrix P exists such that $A^\intercal P + PA \preceq 0$. In discrete-time setting over a discrete linear system $x_{k+1} = Ax_k$, it is defined as

$$\begin{cases} \exists P \succeq 0 \\ A^\intercal PA - P \preceq 0 \end{cases} \tag{7}$$

The LYAPUNOV function $x \mapsto x^\intercal Px$ acts as a measure of energy of the system. When *measuring* the energy of the image state Ax, we obtain $(Ax)^\intercal P(Ax) = x^\intercal A^\intercal PAx$.

Since P is positive definite, $\forall x \in \mathbb{R}^n, x^\intercal Px > 0$, and P denotes a norm over states. While, thanks to the second constraint, its sublevel sets, ie. sets defined by a scalar λ as $x^\intercal Px \leq \lambda$, are inductive over states: $\forall x \in \mathbb{R}^n, x^\intercal Px \geq x^\intercal A^\intercal PAx$. The inequality $A^\intercal PA - P \preceq 0$ encodes a kind of energy dissipation along trajectories.

Therefore these Lyapunov functions describe naturally inductive sets and are good candidate for over-approximation.

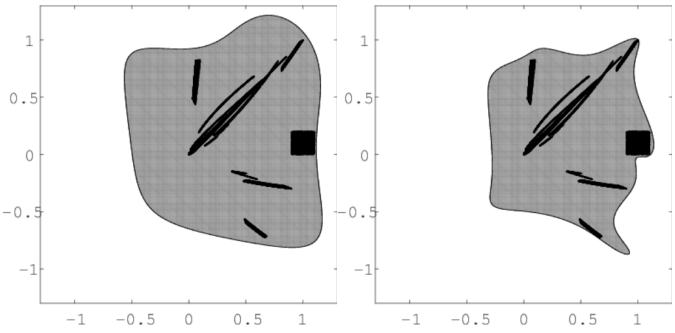
C. Convex-optimization based analysis: automatic synthesis of semialgebraic sets

Ineq. (7) is a Linear Matrix Inequality (LMI). With the development of interior point algorithms [27] and convex optimization [9], the numerical resolution of these convex optimization problems (LMI or SOS) is now feasible in reasonable time. Solving equations such as Eq. (7) produces a inductive sublevel set property for the model semantics.

Let us assume, without loss of generality, that the relationship defined by the predicate $\texttt{step}[i, o, s, s']$ between its internal state before and after the step can be represented by a polynomial function (possibly non-deterministic or even piecewise-defined) s' such that $s' = step(i, s)$ where $step$ is polynomial in i and s.

We can reformulate the definition of the collecting semantics approximation. Let C be a solution of Equation (5),then

$$\begin{cases} \{s \mid \texttt{init}[\texttt{s}]\} \subseteq C\,, \\ \{step(i, s) \mid \exists i \in \mathcal{M}^{I_N}, s \in C\} \subseteq C\,. \end{cases} \tag{8}$$



Black points represent random traces, starting from the black square. Grey regions are computed semialgebraic sets (deg. 8 and 10, resp).

Fig. 2: Computed invariants: inductive semialgebraic sets. [1]

Encoding the set C as the 0-sublevel set of a polynomial p, we obtain the following problem:

$$\begin{cases} \forall s \in \mathcal{M}^{S_N} \text{ s.t. } \texttt{init}[s] & p(s) \leq 0 \\ \forall i \in \mathcal{M}^{I_N}, s \in \mathcal{M}^{S_N} & p\left(step(i, s)\right) \leq p(s)\,. \end{cases} \tag{9}$$

Remark 2. *Note that the constant part of p can be normalized to -1. Then the first constraint rewrites as $p_{nc}(x) \leq 1$ where p_{nc} denotes the non constant part of $p = p_{nc} - 1$. These constants cancel in the second equation.*

This problem is linear in p. It can be solved with an SDP solver using an SOS encoding (cf. [1]). As an example, Fig 2 illustrates computed inductive sublevel set invariants of a piecewise polynomial system: semialgebraic sets. The degrees of the polynomials p are, respectively, 8 and 10.

V. COMPILATION OF DATAFLOW MODELS AND SPECIFICATION

While first compilation schemes for Lustre computed a global automaton of the system [11], the approach of [7] relies on an object-like compilation of the program: each Lustre node call is seen as an instance of the generic declaration of the node. In this compilation scheme, traceability is also more tractable due to the preservation of the program structure. Such compilation is done through three phases: *(i)* First, equations of each Lustre node are transformed in order to extract the stateful computations that appear inside the expressions. Stateful computation can either be the explicit use of a *pre* construct or the call to another node which may be stateful. The extraction is made through a linear traversal of the node's equations, introducing new equations for stateful computation. *(ii)* In a second stage, the set of equations are ordered in order to enable compilation as imperative code. *(iii)* Finally, the C code is generated. Each node instance (object) is represented by a *struct*, which defines its state. In Lustre, syntactic restrictions ensure that recursive definitions are not allowed, therefore side effects are only allowed in identified memories and are updated at each execution of the node body.

The compilation of a Lustre node (I_N, O_N, L_N, Eq_N) will then produce the following C artefacts. The **struct** mem_N describes hierarchically the internal state of nodes, ie. the set S_N.

```
struct mem_N { struct N_reg { ... } _reg;
               struct callee_mem ni_1; ... };
void N_init (struct mem_N *self) { ... };
void N_step (type in, type *out, struct mem_N *self) {..};
```

This compilation scheme can be adapted to produce other outputs. For instance, in the present case, we will produce ACSL predicates, ie. propositional formulas, encoding the transition relation between input stream, output stream and the evolution of the internal state of each node.

The input model components are partitioned into categories: some components are regular model components that will eventually become embedded C code; some others are specification artifacts, synchronous observers, that have to be compiled to ACSL predicates. Since synchronous observers are regular dataflow components, they can be made as complex as possible, for example embedding the (discrete) plant semantics when expressing closed loop properties.

Once both the code and its ACSL specification are produced we obtain the following annotated code (implementing Eq (2)):

```
//@ requires Pre(*self) && Post(*self);
//@ ensures Post(*self);
void N_step (type in, type *out, struct mem_N *self) {..};
```

Formal verification. The generated function contracts can be used to perform verification. When compiled as online checks, these contracts act as test oracles: any use of the functions that violates the contracts will produce an error at runtime. The formal specification can also be used to perform exhaustive analysis using deductive methods. These analyses, as implemented in the WP plugin of Frama-C [15], are similar to the SMT model-checking algorithms: they encode the C semantics as a set of predicates and perform reasoning about this first-order encoding. The encoding is however more complex since it has to formally and soundly represent the memory model of the program (stack of calls, pointers on the heap).

In practice this additional encoding of the memory has a huge impact on the provability of the requirements and, in our experiments, deductive methods rarely succeed in proving these annotations that were previously proved at model level by SMT-based model-checking.

VI. Proof of Refinement

At this stage, the model semantics is both expressed as a set of C functions and as a set of ACSL predicates. Both express the same computations but will produce different encodings when expressed as proof objective.

Consider for example a simple Lustre program accumulating the absolute value of its input i: o = 0 → **pre** o + (**if** i <0 **then** −i **else** i);.

Listing VI.1 presents an ACSL encoding of the node semantics, expressed as a predicate `step` between input, previous and next states of memory. Besides its natural contract `step` one can also define a specification of the C function foo_step denoting the node semantics: a post-condition `nneg` expressing that the updated memory is non-negative.

```
//@ predicate step(i, o, memi, memo) = o == memo &&
(i<0 && memo == memi − i) || (i ≥ 0 && memo == memi + i);
//@ predicate nneg(mem) = mem ≥ 0;
//@ ensures step(i, *o, \at(*self._reg.pre_o, Pre),
                         *self._reg.pre_o);
//@ ensures nneg(*self._reg.pre_o);
void foo_step(int i, int *o, struct mem_foo *self) { ... }
```

Listing VI.1: A first example in ACSL.

In this very simple example, the proof objective generated when considering the *ensures* clauses will contain all the axiomatization of the memory model and will include arrays/indices representing memory/pointers, while a theorem stating that the property *nneg* is inductive with respect to *step* would generate a much simpler proof objective. Note the use of the ACSL function *at* which allows to access to the value of the pointer before the call to the function.

As a consequence, it is much more reliable to perform the proof in two stages: i) first, prove the inductiveness of the property with respect to an predicate encoding in ACSL of the model semantics and ii) prove that the C function do conform to the relationship expressed by the ACSL predicate describing the semantics of the model. This second challenge amounts to performing a refinement proof of the ACSL predicate by the C function.

We present here our method to automatize this refinement proof through the definition of intermediate ACSL predicates describing each computation step.

A. Memory Representation

We first map the low-level memory representation to a simpler model, more suited to verification purposes. We propose to build a *ghost* memory representation in ACSL, with the same structure but without any pointer, using only structs within structs. For a node N, we then consider a simulation relation $\texttt{ghost}_N$, pictured in Figure 3, between real (mem_N) and ghost (mem_ghost_N) memory, which equates corresponding struct fields of both memories, disregarding pointers. The stateful operator "−>" also has its own ghost memory. Listing VI.2 shows the requirements that the N_step function must meet.

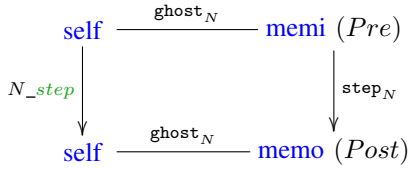


Fig. 3: The ghost simulation relation.

Besides, the validity of the C memory state must be ensured through the ACSL predicate separated . A valid memory has all its pointers allocated and pairwise different. Similar requirements are needed for node initialization, both for the C function and the ACSL predicate.

B. Code Annotation and Optimization

Finally, after every equation processing, depending whether a node memory is assigned, a different simulation relation

```
/*@ requires ...; /* memory validity predicate */
    ensures \forall struct mem_ghost_N memi;
            \forall struct mem_ghost_N memo;
            \at(ghost_N(memi, self), Pre)
             ==> ghost_N(memo, self)
             ==> step_N(i_1, ..., i_a, *o_1, ..., *o_c,
                        memi, memo);
    assigns ...; /* assigned memory and output */*/
void N_step (I_1 i_1, ..., I_a i_a,
             O_1 (*o_1), ..., O_c (*o_c),
             struct mem_N *self) {...}
```

Listing VI.2: Code contracts with ghost memories in ACSL.

$\texttt{ghost}_N^k$ and annotation $\texttt{step}_N^k$ has to be provided, still following the preceding general scheme of Listing VI.2. These relations account for the different partial matchings between initial and updated memory variables. We also existentially quantify over free local variables as soon as they are not *live* anymore, according to a liveness analysis. It allows to activate various optimizations (such as equation inlining and non-*live* variable reuse) during code generation, without compromising the proof process. For the sake of simplicity, we elude here the precise treatment of these annotations. The resulting predicates $\texttt{step}_N^i$ are defined in Figure VI.3. $Locals$ and $Live^i$ respectively denote the set of local and live variables after the evaluation of equation eq^i. $\texttt{ACSL\_}eq^i$ denotes the translation of equation i to ACSL, as in Figure VI.1.

```
/*@ predicate
step_N^{i+1}(I_1 i_1 ... i_a, Locals ∩ Live^{i+1}, Outputs ∩ Live^{i+1},
     struct mem_ghost_N memi, struct mem_ghost_N memo) =
\exists Locals ∩ (Live^i \ Live^{i+1});
  step_N^i(i_1, ..., i_a, Locals ∩ Live^i, Outputs ∩ Live^i,
           memi, memo)
  && ACSL_eq^{i+1};*/
void N_step (...) {
  eq^i; /* after each equation */
  /*@ assert \forall struct mem_ghost_N memi;
             \forall struct mem_ghost_N memo;
             \at(ghost_N(memi, self), Pre)
              ==> ghost_N^i(memo, self)
              ==> step_N^i(...);*/
  ...}
```

Listing VI.3: Annotation predicates in ACSL.

VII. Validating numerical invariants

As presented in Sec. IV invariants are mandatory to reinforce the semantics description and enable the validation of requirements. Such invariants were computed at model level and can be expressed at code level. However two main issues arise: first, these numerical invariants are typically non linear, from simple quadratic sets to more general semi-algebraic sets (sublevel set of polynomials), and most SMT-solvers have limited capabilities to manipulate predicates over non-linear arithmetics; second, all numerical computation were assumed to be performed with Reals, while, in practice they will be implemented with imprecise machine type representations such as floating point arithmetics. We present here how we propose to address these issues, in a global formal framework.

A. SMT-solver with support of polynomial real arithmetics

To revalidate the invariants computed in Section IV, SMT solvers will face goals such as

$$\begin{aligned} &\forall x, init(x) \leq 0 \implies p(x) \leq 0 \\ &\forall x, p(x) \leq 0 \implies p(step(x)) \leq 0 \end{aligned} \tag{10}$$

where p, $init$ and $step$ are given polynomials. As already stated, these will be adressed by attempting to prove that the following formulas are unsatisfiable

$$\begin{aligned} &init(x) \leq 0 \wedge p(x) > 0 \\ &p(x) \leq 0 \wedge p(step(x)) > 0. \end{aligned} \tag{11}$$

The theory of polynomial real arithmetic is decidable but, since the best algorithms remain costly, state of the art SMT-solvers rely on various heuristics to attempt to perform these proofs. Unfortunately, they often fail to prove the above goals.

A sufficient condition for (10) is the existence of polynomials $\sigma_1, \ldots, \sigma_4$ such that $\sigma_1 \geq 0, \ldots, \sigma_4 \geq 0$ and

$$\begin{aligned} &\sigma_1\, init - \sigma_2\, p > 0 \\ &\sigma_3\, p - \sigma_4\, p(step) > 0. \end{aligned}$$

For an arbitrary bounded degree, such polynomials can be looked for using a SOS encoding and a SDP solver. Unfortunately, due to the nature of the interior point algorithms they implement, these solvers provide approximate numerical results that may be incorrect and must be checked. This check amounts to checking that some matrix is positive definite, which can efficiently be performed using a floating-point Cholesky decomposition and carefully bounding its rounding errors [35]. For increased confidence, this algorithm has been formally proved [32] using the Coq proof assistant.

Implementing the above procedure in a SMT-solver enables to revalidate invariants computed in Section IV [33].

B. Addressing floating point imprecision

Actual implementations will likely use some finite precision arithmetic such as floating-point arithmetic. Thus, one must check that the invariant previously computed assuming arithmetic computations in the real field are still valid for the floating-point implementation.

Controllers are usually designed to offer some robustness qualities. They are for instance able to withstand some amount of imprecision in the input sensors or output actuators. It is thus expected that they also withstand rounding errors induced by the floating-point implementation, considering that the latter are commonly of a much smaller magnitude. This however remains to be checked.

A simple solution is to use the standard model of floating-point arithmetic to represent each floating-point operation as the corresponding one in the real field plus some small error ϵ. These errors can then be accumulated and hopefully, it can be proved that the invariant remain inductive despite the additional error. Again, for quadratic invariants on linear systems

(the most common case), the details have been proved [32] using the proof assistant Coq. This additional guarantee is particularly welcome for such proofs which are typically rather easy but tedious, hence error prone.

VIII. Conclusion

We presented a framework allowing to support translation validation, while dealing with functional specification. The proposed approach supports formal specification and formal verification all along the software development cycle, from model design while manipulating dataflow models, to code level verification and validation.

The approach amounts to adapt a compilation process to compile specification and proof, and to support their later re-validation.

In terms of perspectives, the first one is a complete implementation of the methodology, linking the existing tools together. We would also like to address the specification of system-level properties (such as stability, robustness or performance) and to consider hybrid closed-loop systems where the plant semantics is defined by ODEs and the controller is a regular (discrete time) dataflow model.

References

[1] A. Adje, P.-L. Garoche, and V. Magron. A sums-of-squares extension of policy iterations. *Nonlinear Analysis: Hybrid Systems*, 2017.

[2] P. Amagbégnon, L. Besnard, and P. L. Guernic. Implementation of the data-flow synchronous language signal. In *PLDI'95*, pages 163–173. ACM, 1995.

[3] K. J. Astrom and R. M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, 2008.

[4] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. ACSL: ANSI/ISO C Specification Language. version 1.7.

[5] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. In *Proc. of the IEEE*, pages 1270–1282, 1991.

[6] G. Berry and G. Gonthier. The esterel synchronous programming language: design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.

[7] D. Biernacki, J.-L. Colaço, G. Hamon, and M. Pouzet. Clock-directed modular code generation for synchronous data-flow languages. In *LCTES*, pages 121–130, 2008.

[8] T. Bourke, L. Brun, P. Dagand, X. Leroy, M. Pouzet, and L. Rieg. A formally verified compiler for lustre. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 586–601, 2017.

[9] S. Boyd and L. Vandenberghe. *Convex optimization*. Cambridge University Press, New York, NY, USA, 2004.

[10] A. Bradley. Understanding IC3. In *SAT 2012*, pages 1–14, 2012.

[11] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre: A declarative language for programming synchronous systems. In *POPL*, pages 178–188, 1987.

[12] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: a declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '87, pages 178–188. ACM, 1987.

[13] CENELEC. *CENELEC 50128 - Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems*, 2001.

[14] D. Cofer, A. Gacek, S. Miller, M. Whalen, B. LaValley, and L. Sha. Compositional verification of architectural models. In *NASA Formal Methods - 4th International Symposium, NFM 2012*, pages 126–140, 2012.

[15] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c: a software analysis perspective. SEFM'12, pages 233–247. Springer, 2012.

[16] DO-178C, software considerations in airborne systems and equipment certification, 2011.

[17] P.-L. Garoche, A. Gurfinkel, and T. Kahsai. Synthesizing modular invariants for synchronous code. In *Proceedings First Workshop on Horn Clauses for Verification and S ynthesis, HCVS 2014, Vienna, Austria, 17 July 2014.*, pages 19–30, 2014.

[18] G. Hagen and C. Tinelli. Scaling up the formal verification of Lustre programs with SMT-based techniques. In *FMCAD-2008*, pages 109–117. IEEE, 2008.

[19] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. In *Proceedings of the IEEE*, pages 1305–1320, 1991.

[20] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In *AMAST*, pages 83–96, 1993.

[21] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[22] N. Izerrouken, X. Thirioux, M. Pantel, and M. Strecker. Certifying an automated code generator using formal tools : Preliminary experiments in the geneauto project. In *European Congress on Embedded Real-Time Software (ERTS), Toulouse, 29/01/2008-01/02/2008*, http://www.sia.fr, 2008. Société des Ingénieurs de l'Automobile.

[23] J. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie. A formally-verified C static analyzer. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT POPL 2015, Mumbai, India, January 15-17, 2015*, pages 247–259, 2015.

[24] T. Kahsai and C. Tinelli. PKIND: a parallel k-induction based model checker. In *PDMC*, volume 72 of *EPTCS*, pages 55–62, 2011.

[25] G. C. Necula. Translation validation for an optimizing compiler. *SIGPLAN Not.*, 35(5):83–94, May 2000.

[26] G. C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 83–94, New York, NY, USA, 2000. ACM.

[27] Y. Nesterov and A. Nemirovski. *Interior-point Polynomial Algorithms in Convex Programming*, volume 13 of *Studies in Applied Mathematics*. Society for Industrial and Applied Mathematics, 1994.

[28] B. Pagano, O. Andrieu, T. Moniot, B. Canou, E. Chailloux, P. Wang, P. Manoury, and J. Colaço. Experience report: using objective caml to develop safety-critical embedded tools in a certification framework. In *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*, pages 215–220, 2009.

[29] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '98, pages 151–166, London, UK, UK, 1998. Springer-Verlag.

[30] A. Rantzer. On the kalman-yakubovich-popov lemma. *Syst. Control Lett.*, 28(1):7–10, June 1996.

[31] A. Rantzer. On the kalman-yakubovich-popov lemma for positive systems. *IEEE Trans. Automat. Contr.*, 61(5):1346–1349, 2016.

[32] P. Roux. Formal proofs of rounding error bounds - with application to an automatic positive definiteness check. *J. Autom. Reasoning*, 2016.

[33] P. Roux, M. Iguernlala, and S. Conchon. A non-linear arithmetic procedure for control-command software verification. In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II*, pages 132–151, 2018.

[34] P. Roux, R. Jobredeaux, and P.-L. Garoche. Closed loop analysis of control command software. In A. Girard and S. Sankaranarayanan, editors, *18th International Conference on Hybrid Systems: Computation and Control (part of CPS Week), HSCC'15, Seattle, Washington, USA, April 14-16, 2015,*, pages 108–117, 2015.

[35] S. M. Rump. Verification of positive definiteness. *BIT Num. Math.*, 2006.

[36] J. Rushby. The versatile synchronous observer. In *SBMF*, pages 1–1. Springer, 2012.

[37] T. Wang, P.-L. Garoche, P. Roux, R. Jobredeaux, and É. Féron. Formal analysis of robustness at model and code level. In *19th International Conference on Hybrid Systems: Computation and Control (part of CPS Week), HSCC'16, Vienna, Austria, April 12-14, 2016*, 2016.

[38] M. Westhead and S. Nadjm-Tehrani. Verification of embedded systems using synchronous observers. In *Conference on Formal Techniques in Real-time and Fault-tolerant Systems, LNCS 1135*, pages 405–419. Springer, 1996.