A Logic Theory Pattern for Linearized Control Systems[*]

Andrea Domenici Cinzia Bernardeschi

Department of Information Engineering
University of Pisa
Pisa, Italy

{andrea.domenici,cinzia.bernardeschi}@unipi.it

This paper describes a procedure that system developers can follow to translate typical mathematical representations of linearized control systems into logic theories. These theories are then used to verify system requirements and find constraints on design parameters, with the support of computer-assisted theorem proving. This method contributes to the integration of formal verification methods into the standard model-driven development processes for control systems. The theories obtained through its application comprise a set of assumptions that the system equations must satisfy, and a translation of the equations into the logic language of the Prototype Verification System theorem-proving environment. The method is illustrated with a standard case study from control theory.

1 Introduction

The design of modern control systems relies both on theory and on experiment. Experiment often takes the form of simulation, in a process known as *model-driven development* (MDD). Both the theory- and the experiment-based activities start from a mathematical system model, but in one case the model is a static subject of formal analysis, and in the other one it is an active system under observation. This implies that developers involved in the two branches of development require different kinds of knowledge and the model itself may have to be expressed with different notations.

The present paper is an initial contribution to a better integration between formal analysis and MDD, as it provides a systematic way to translate a mathematical system description into a logic theory that can be analyzed with the support of an interactive theorem prover. The proposed procedure is tailored to basic linearized control systems and it follows a high-level pattern, so that its application requires some creativity on the part of the developers. The procedure supports developers in providing them with a framework upon which to build system-specific theories. Parts of the procedure can be embodied into code templates for specific classes of systems. This work also tries to contribute in adding support for control system development to the large body of theories for the Prototype Verification System (PVS) [4]. In particular, a solution for a technical problem related to the PVS type system is proposed (Sect. 4.1 and 4.2).

The PVS environment is introduced in Sect. 2, Sect. 3 describes the system used as an example, Sect. 4 describes the procedure within the example, and the procedure is exposed in general terms in Sect. 5. Conclusions are drawn in Sect. 6.

2 The PVS theorem-proving environment

The PVS is an interactive theorem prover for theories in higher-order logic. Theories are written in a rich, strongly-typed language. In this paper, only the features needed to understand the examples are

[*] This research was funded by the Italian Government under the CrossLab program, “Dipartimenti di Eccellenza” project.

A. Paskevich, J. Proença (Eds.): Formal Integrated
Development Environment 2021 (F-IDE 2021)
EPTCS 338, 2021, pp. 46–52, doi:10.4204/EPTCS.338.7

introduced.

The type system rests on the fundamental concepts of *reals*, *rationals*, *integers*, etc., whose properties and operations are formally defined in the built-in theories of the prover. Subtypes (e.g., `posreal`, positive reals) and types of arbitrary complexity can be defined upon this basis. In particular, *function types* are defined as in this example:

```
phi: VAR [real -> real]
```

where `phi` is defined as a *function variable* ranging over the set of functions from reals to reals. A *function constant* (not to be confused with a constant function) is defined, e.g., as

```
x: VAR real
f(x): real = -(g/R)*sin(x)
```

Formulae are made of *predicates* (functions returning a Boolean value), logical connectives, and quantifiers. A `LET ... IN` clause introduces a definition into the following expression. A `COND` expression selects one value from a set, associated with one of a set of mutually exclusive Boolean expressions.

Several theories from the NASALIB package [3] are used in this work, in particular, those related to mathematical analysis and vectors. Type `Vect2` represents the set of real 2-vectors. Function `vect2` is a constructor, and `p(0)` (or `p(1)`) is the first (second) element of a vector `p`.

Logical statements to be proved are expressed as logic formulae introduced by a label and a keyword such as `LEMMA` or `THEOREM`:

```
label: LEMMA f(phi, theta) = g(theta)
```

In the interactive theorem prover, the user selects one lemma as the initial goal and applies prover commands to transform it. The commands are based on the inference rules of sequent calculus [6] and perform transformations of various complexity. In particular, they may recursively split a goal into subgoals. A proof terminates successfully when all subgoals are satisfied.

3 Equilibrium of a pendulum

A rigid pendulum with a fixed pivot is a simple example of a second-order dynamical system [5]. A pendulum consists in a bob of mass M and a rod of length R. The pivot has a coefficient b of viscous friction. The pendulum is controlled with a driving torque τ applied at the pivot. The dynamic model of the system is then

$$\mathrm{I}\ddot{\theta} + \mathrm{b}\dot{\theta} + \mathrm{MgR}\sin(\theta) = \tau \,,$$

where θ is the angular displacement from the vertical axis, $\mathrm{I} = \mathrm{MR}^2$ is the rotational inertia, and g is the gravitational acceleration. Taking θ and $\dot{\theta}$ as the state variables, the state-space representation for the free evolution of the system is

$$\begin{cases} \dot{x_1} = \dot{\theta} \\ \dot{x_2} = -(\mathrm{g}/\mathrm{R})\sin(\theta) - (\mathrm{b}/\mathrm{I})\dot{\theta} \,, \end{cases}$$

where the right-hand members of the equalities are the system's *generating functions* (GF). The equilibrium points are $\mathrm{P}_1 = (0,0)$ and $\mathrm{P}_2 = (\pi,0)$.

4 A logic theory

This section shows how a theory can be developed for the formal study of a dynamical system such as the one introduced above.

In a standard approach, the system is linearized near an equilibrium point, i.e., its GFs (f_1 and f_2) are replaced by their differentials. The linearized system is expressed as

$$\dot{\mathbf{x}} = \mathbf{J}\mathbf{x}\,,$$

where $\mathbf{x}$ is the state vector and the Jacobian matrix $\mathbf{J}$ is defined as

$$\mathbf{J} = \begin{bmatrix} \dfrac{\partial f_1}{\partial \theta} & \dfrac{\partial f_1}{\partial \dot{\theta}} \\ \dfrac{\partial f_2}{\partial \theta} & \dfrac{\partial f_2}{\partial \dot{\theta}} \end{bmatrix}\,.$$

4.1 System model

The first step in building the system's theory is then to express the GFs and their partial derivatives (PD) in the PVS language. Translating mathematical formulae into PVS is obviously straightforward, but the theory must provide the means to verify the correctness of the procedure. In particular, it is necessary to verify that the functions are actually differentiable, and that the derivatives have been calculated correctly. Dealing with these issues leads to coping with some limitations in the available PVS libraries. In fact, to the best of the authors' knowledge, there are currently no PVS theories available about partial derivation. However, a rich collection of theories about functions of a single variable is available, which can be used as a basis for a treatment of multi-variable functions.

This work takes a pragmatic approach: Instead of trying to build a general theory of multi-variable functions, *ad hoc* theories are developed for specific systems, trying to strike a satisfactory compromise between generality (and elegance) and synthesis.

From calculus textbooks, the PD of a function $f : \mathbb{R}^n \to \mathbb{R}$ wrt a variable x_i in a point $\mathrm{P}_0 = (x_{01}, \ldots, x_{0n})$ is the ordinary derivative of the *restriction* $\phi : \mathbb{R} \to \mathbb{R}$ of f to the set $\{x_{01}, \ldots, x_i, \ldots, x_{0n}\}$. This is usually translated as "*take the derivative of f wrt x_i, keeping the other variables constant*". Unfortunately, the PVS type checker "does not know" how to keep variables constant: A variable is supposed to be variable and a constant is supposed to be constant. More precisely, it is not possible to treat a multi-variable function as a single-variable one, take its single-variable derivative and turn it into a multi-variable function.

In the approach proposed in this work, a developer must write the GFs, the points where the system is linearized, the single-variable restrictions, the (ordinary) derivatives of the restrictions, and the PDs of the GFs. With an explicit definition of the single-variable restrictions and their derivatives, it is possible to check the correctness of the "hidden step" in the common calculation of PDs. The pendulum system can then be defined as in the following. First, parameters, variables, GFs, and equilibrium points:

```
M, R, b: posreal        % mass, length, viscous friction
g: posreal              % (standard) gravitational acceleration
I: posreal = M*sq(R)    % rotational inertia
tau: VAR real           % applied torque
theta, dtheta: VAR real % (state variables) angular displmt., time derivative

% generating functions
p: VAR Vect2            % a generic point p = (theta, dtheta)
f1(p): real = p(1)                          % p(1) = dtheta
f2(p): real = -(g/R)*sin(p(0)) - (b/I)*p(1)  % p(0) = theta

% equilibrium points
```

```
P1: Vect2 = vect2(0, 0);  X1: real = P1(0); Y1: real = P1(1)
P2: Vect2 = vect2(pi, 0); X2: real = P2(0); Y2: real = P2(1)
```

Then the restrictions are defined by applying the GFs to points with one variable and one arbitrary constant co-ordinate, and their derivatives are computed:

```
X, Y: real
P: Vect2 = vect2(X, Y)                       % an arbitrary constant point
phi1_x(theta): real = f1(vect2(theta, Y))    % restr. of f1 wrt theta
phi1_y(dtheta): real = f1(vect2(X, dtheta))  %        "        dtheta
phi2_x(theta): real = f2(vect2(theta, Y))    % restr. of f2 wrt theta
phi2_y(dtheta): real = f2(vect2(X, dtheta))  %        "        dtheta

% derivatives of the restrictions
dphi1_x(theta): real = 0;                 dphi1_y(dtheta): real = 1
dphi2_x(theta): real = -(g/R)*cos(theta); dphi2_y(dtheta): real = -(b/I)
```

Finally, the PDs of the GFs are declared:

```
df1_dx(p): real = 0                   % partial derivative of f1 wrt theta
df1_dy(p): real = 1                   % partial derivative of f1 wrt dtheta
df2_dx(p): real = -(g/R)*cos(p(0))    % partial derivative of f2 wrt theta
df2_dy(p): real = -(b/I)              % partial derivative of f2 wrt dtheta
```

By convention, letters x and y refer to the first and second state variable (θ and $\dot{\theta}$), respectively.

4.2 Model consistency

In order to verify the correctness of ordinary differentiations, the NASALIB theories are used. For example, the following lemma states that the `dphi1_x` is indeed the derivative of `phi1_x` wrt `theta`, using function `deriv` defined in the NASALIB `derivative` theory:

```
der1theta: LEMMA deriv(phi1_x, theta) = dphi1_x(theta)
```

Then the PDs must be proved to be extensions of the derivatives of the GFs' restrictions. Predicates `xrestricts?` check if a function $\phi : \mathbb{R} \to \mathbb{R}$ is a restriction of a function $f : \mathbb{R}^2 \to \mathbb{R}$ (and conversely).

```
x, y: VAR real
f: VAR [Vect2 -> real]        % a generic f: (R x R) -> R
phi: VAR [real -> real]       % its restriction phi: R -> R

xrestricts?(phi, f, p): bool = FORALL (x): phi(x) = f(vect2(x, p(1)))
yrestricts?(phi, f, p): bool = FORALL (y): phi(y) = f(vect2(p(0), y))
```

These predicates are then used first to show that `phi1_x` etc. are restrictions of the respective GFs, and then that the PDs `df1_dx` etc. are extensions of the derivatives of the restrictions. For example:

```
restr1x1: LEMMA xrestricts?(phi1_x, f1, P)
extens1: LEMMA
    xrestricts?(dphi1_x, df1_dx, P) AND yrestricts?(dphi1_y, df1_dy, P)
```

After the consistency of the various functions written by the developer has been verified, the task remains to check that linearizing the system at the equilibrium points is mathematically sound, i.e., that the GFs are differentiable. Elementary calculus provides a sufficient condition for differentiability: "*If f has a partial derivative in a point P_0 and the other partial derivatives exist in a ball around P_0 and are continuous in P_0, then f is differentiable in P_0*". These conditions can be checked with NASALIB theories on ordinary differentiation and on continuity in Euclidean spaces. e.g.:

```
% does a partial derivative of f1 exist in P1?
exist_pd1P1?: bool = derivable?(phi1_x, X1) OR derivable?(phi1_y, Y1)

% is the partial derivative of f1 continuous at P1?
continuous_pdx1P1?(S: VAR set[Vect2]): bool = continuous_at?(df1_dx, X1, P1)
```

4.3 Linearization

Finally, the Jacobian and its trace and determinant can be defined at the points of equilibrium. The *matrices* NASALIB theory was not used as deemed unnecessarily complex.

```
JP1(i, j: below(2)): real =                              % Jacobian at P1
    LET idx = 3*(i - 1) + j IN COND
        idx = 1 -> df1_dx(P1), idx = 2 -> df1_dy(P1),
        idx = 3 -> df2_dx(P1), idx = 4 -> df2_dy(P1)
    ENDCOND

trJP1: real = JP1(1, 1) + JP1(2, 2)                      % trace of JP1

detJP1: real = JP1(1, 1)*JP1(2, 2) - JP1(1, 2)*JP1(2, 1) % determinant of JP1
```

Then, the characteristic polynomial, its discriminant, and the eigenvalues:

```
lam: VAR complex
csq(lam): complex = lam*lam    % complex square
charpolJP1(lam): complex = csq(lam) - trJP1*lam + detJP1

% discriminant
discrJP1: real = discr(1, -trJP1, detJP1)

% eigenvalues
lam1: complex =
    IF (discrJP1 >= 0) THEN root(1, -trJP1, detJP1, -1)
    ELSE trJP1/2 - i*sqrt(-discrJP1)/2
    ENDIF
lam2: complex =
    IF (discrJP1 >= 0) THEN root(1, -trJP1, detJP1, 1)
    ELSE trJP1/2 + i*sqrt(-discrJP1)/2
    ENDIF
```

This code has a regular structure and has very few dependencies on the developer-provided code, so it could be easily turned into a template.

4.4 Analysis

Within this theory, it is possible to prove properties of the system, and, in particular, to find constraints on the physical parameters. For example, it has been proved that the system has non-oscillating solutions in P_1 (i.e., the discriminant of the characteristic polynomial is positive) if and only if the ratio of viscous friction to rotational inertia is greater than four:

```
lem4: LEMMA K = b/I AND K > 4 IFF discrJP1 > 0
```

The proof with the PVS proof assistant is straightforward, involving only simple manipulations, such as introducing previously proved lemmas and expanding definitions. Similarly, it has been proved that P_2 is unstable.

5 General procedure

The above process can be described in a more general and schematic procedure as shown below. It may be noted that, after a developer has written the system model in PVS, most of the remaining theory, e.g., the lemmas in Sec. 4.2 and the definitions in Sec. 4.3, is a set of boilerplate definitions that could be generated with a template-processing software.

System model
The developer defines (a) the *state variables* $x_1, \dots, x_n$ (`theta` and `dtheta` in Sec. 4.1), (b) the *system parameters* $k_1, \dots, k_l$ (as constants, e.g., `M`, `R`, `b`, `g`), (c) the system *generating functions* $f_1, \dots, f_n$ (`f1`, `f2`), (d) the *equilibrium points* (`P1`, `P2`), (e) the *restrictions* $\phi_{11}, \dots, \phi_{nn}$ (`phi1_x`, ...`phi2_y`) of the GFs, (f) the *derivatives* $\phi'_{11}, \dots, \phi'_{nn}$ (`dphi1_x`, ...`dphi2_y`) *of the restrictions*, and (g) the *partial derivatives* $\frac{\partial f_1}{\partial x_1}, \dots, \frac{\partial f_n}{\partial x_n}$ (`df1_dx`, ...`df2_dy`) of the GFs.

Model consistency
The theory contains predicates and lemmas to check that (a) the ϕ_{ij}'s are actually the restrictions of the f_i's, (b) the ϕ'_{ij}'s are actually the derivatives of the ϕ_{ij}'s, (c) the $\frac{\partial f_i}{\partial x_j}$'s are extensions of the ϕ_{ij}'s, and (d) the generating functions are differentiable, as shown in Sec. 4.2. In this phase, lemmas such as `restr1x1` and `extens1` formalize the concept of PD in a way that is acceptable to a higher-order logic type checker.

Linearization
For each equilibrium point, write (a) the *Jacobian* matrix (`JP1` in Sec. 4.3), (b) the functions of the Jacobian needed to write the characteristic polynomial, e.g., *trace* and *determinant* (`trJP1` and `detJP1`) for second-order systems), (c) the characteristic polynomial (`charpolJP1`), (d) functions of the polynomial needed to characterize the set of eigenvalues, e.g., the *discriminant* (`discrJP1`) for second-order systems), and (e) the expressions of the eigenvalues (`lam1`, `lam2`).

Analysis
Use the functions from the linearization phase to write lemmas about system properties, e.g., Lemma `lem4` in Sec. 4.4, relating stability to parameter ranges. In real life applications, this part will require most of the total effort. It should be observed that a PVS theory need not be monolithic, so that the *divide and conquer* principle can and should applied to cope with problem size and complexity. Also, formal analysis requires specific expertise, but a systems engineer can easily learn the essentials to define the system model and its requirements, leaving theorem proving to specialized developers (and their software).

6 Conclusions and further work

The procedure proposed in this paper had been used, before being laid down explicitly, in the analysis of a simple robotic vehicle [2] and of a synchronous motor [1]. In both cases, formal verification was complemented by simulation, and in the latter, by design space analysis. More precisely, formal verification had been used to find useful ranges of controller gains and design space analysis, supported by the simulation environment, was used to find optimal values within those ranges. The present work sketches a systematic way to deal with this kind of tasks. Clearly, the procedure shown in Sect. 5 needs to be defined more in detail and templates for theory fragments have to be defined. Also, the analysis of systems with a large state space will require more advanced strategies to develop appropriate theories.

References

[1] Cinzia Bernardeschi, Pierpaolo Dini, Andrea Domenici, Maurizio Palmieri & Sergio Saponara (2020): *Formal Verification and Co-Simulation in the Design of a Synchronous Motor Control Algorithm*. Energies 13(16), p. 4057, doi:10.3390/en13164057.

[2] Andrea Domenici, Adriano Fagiolini & Maurizio Palmieri (2018): *Integrated Simulation and Formal Verification of a Simple Autonomous Vehicle*. In Antonio Cerone & Marco Roveri, editors: *Software Engineering and Formal Methods*, *Lecture Notes in Computer Science* 10729, Springer International Publishing, Cham, pp. 300–314, doi:`10.1007/978-3-319-74781-1_21`.

[3] Bruno Dutertre (1996): *Elements of mathematical analysis in PVS*. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Joakim von Wright, Jim Grundy & John Harrison, editors: *Theorem Proving in Higher Order Logics*, *Lecture Notes in Computer Science* 1125, Springer Berlin Heidelberg, pp. 141–156, doi:`10.1007/BFb0105402`.

[4] S. Owre, S. Rajan, J. Rushby, N. Shankar & M. Srivas (1996): *PVS: combining specification, proof checking, and model checking*. In R. Alur & T.A. Henzinger, editors: *Computer-Aided Verification, CAV '96*, *LNCS* 1102, Springer-Verlag, pp. 411–414, doi:`10.1007/3-540-61474-5_91`.

[5] J. E. Slotine & W. Li (1991): *Applied Nonlinear Control*. Prentice-Hall, Englewood Cliffs, NJ.

[6] Raymond Merrill Smullyan (1968): *First-order logic*. Ergebnisse der Mathematik und ihrer Grenzgebiete, Springer, Berlin, Heidelberg, doi:`10.1007/978-3-642-86718-7`.