Integrated simulation and formal verification of a simple autonomous vehicle

A. Domenici[1], A. Fagiolini[2], and M. Palmieri[3 1]

[1] Dept. of Information Engineering, University of Pisa
[2] Dipartimento di Energia, Ingegneria dell'Informazione e Modelli Matematici (DEIM), University of Palermo
[3] DINFO, University of Florence

Abstract. This paper presents a proof-of-concept application of an approach to system development based on the integration of formal verification and co-simulation. A simple autonomous vehicle has the task of reaching an assigned straight path and then follow it, and it can be controlled by varying its turning speed. The correctness of the proposed control law has been formalized and verified by interactive theorem proving with the Prototype Verification System. Concurrently, the system has been co-simulated using the Prototype Verification System and the MathWorks Simulink tool: The vehicle kinematics have been simulated in Simulink, whereas the controller has been modeled in the logic language of the Prototype Verification System and simulated with the interpreter for the same language available in the theorem proving environment. With this approach, co-simulation and formal verification corroborate each other, thus strengthening developers' confidence in their analysis.

1 Introduction

Simulation and formal verification are complementary techniques, both required in the development of complex, possibly safety-critical systems. Formal specification enables developers to deal with complexity using well-proven tools of logic and mathematics, providing strong assurance on compliance with requirements. On the other hand, it is always possible to correctly formalize wrong assumptions, or to prove wrong conclusions from wrong assumptions. It is also possible to produce simply wrong proofs, but this risk is mitigated by the use of automatic or interactive theorem proving. This given, simulation provides sanity checks at early stages of development, besides being a prototyping tool supporting the exploration of user interaction.

In the field of CPSs, simulation often takes the form of *co-simulation*, i.e., integrated simulation of different subsystems, each modeled with a specific formalism and simulated by a specific simulation engine. The need for co-simulation arises naturally from the fact that CPSs are usually composed of parts that follow different physical laws, or must be described under different aspects: For example, the rotor, stator, and winding of an electric motor are both electrical and mechanical systems.

A further motivation for using co-simulation stems from the previous considerations on the complementarity of simulation and verification, and also from the separation of controller and plant in a CPS: A model of the controller expressed in a logic language can be proved correct with respect to a model of the plant expressed in the same language, then the controller model can be simulated using an interpreter for that language, along with a simulation of a plant model built with an application-specific formalism, such as, e.g., a Simulink toolbox.

This paper illustrates the above approach to integrated co-simulation and verification with a simple case study from the field of autonomous vehicles. The case study concerns a single-axle vehicle, which moves at constant speed and whose turning speed can be controlled. The controller must be able to steer the vehicle until it reaches its assigned path, a straight line. The kinematics of the vehicle and the control law have been expressed in the higher-order logic language of the Prototype Verification System (PVS) [22]. Using the well-established methods of control theory, it has been proved that the target configuration (i.e., the vehicle following an assigned straight line) is an asymptotically stable state. Concurrently, a Simulink model of the vehicle's kinematics has been co-simulated with the PVS specification of the control law.

In the rest of the paper, Section 2 cites work related to the topics of this paper; Section 3 provides basic information on the PVS environment; Section 4 describes the case study; Section 5 reports on the verification of the considered system; Section 6 reports on its co-simulation; and Section 7 concludes the paper.

2 Related Work

Work on co-simulation of CPSs has produced a large body of literature that cannot be surveyed exhaustively within the limits of the present paper. Only a small number of recent works will be cited, while the reader is referred to more extensive reviews, such as [11].

The Vienna Development Method (VDM) [9] and the Bond-Graph notation [13] have been used in the Crescendo tool [15] to co-simulate discrete systems in VDM and continuous systems with Bond-Graphs.

In the approach proposed by Attarzadeh *et al.* [20], heterogeneous processes, executing models expressed in different modeling or programming languages, or even implemented in hardware, are organized hierarchically and coordinated by a framework that takes into account each process's *Model of Computation* [16] defining the time, synchronization, and communication models of the process.

Differential dynamic logic [24] is used with the KeYmaera X [10] theorem prover, developed for the verification of CPSs. Its language includes conditions, non-determinism, loops, composition, and continuous dynamics, i.e., behaviors defined by differential equations.

Another family of operational-style formalisms that produce executable models is the one of languages based on Petri Nets, such as *Stochastic Activity Networks* [25], used, e.g., to model FPGAs [1, 2].

The PVS environment has been used in several application fields, such as hardware verification [23] or air traffic control [6]. Recently, it has been used to verify the specification and the implementation of a set of collision-avoidance algorithms for unmanned aircraft systems [19]. The authors of the present paper used the PVS environment to co-simulate an implantable pacemaker with a Simulink model of the heart [4]. The pacemaker was modeled by a PVS theory as a network of timed automata and executed in the PVSio-web framework [21, 17], connected to a Simulink tool executing the heart model. The PVS language has also been used to verify a simple nonlinear hybrid control system [3].

3 Background on the PVS Environment

The PVS theorem prover is based on higher-order logic and the sequent calculus [26]. A PVS user writes *theories* containing definitions of types, constants, and variables. A function is a constant whose type is the function's signature. For example, the type expression '$[int \to real]$' denotes the type of functions from integers to reals. Variables may range over function types, and a function type may have other function types as domain and codomain.

A theory also contains statements of two kinds: *axioms*, assumed as valid by the prover, and *theorems* to be proved. A proved theorem can be used in further proofs. The language identifies statements with labels followed by the keywords *axiom* for axioms and *theorem*, *lemma*, or others, for theorems.

A theory may refer to other theories introduced by the *importing* declaration. A large number of pre-proved fundamental theories is collected in the *prelude* library and implicitly imported in all theories. Many other theories are available in several libraries, such as the NASA PVS Library (NASALIB) [8, 12].

The PVS deduction system is based on sequent calculus. A *sequent* is an expression of the form $A_1, A_2, \ldots, A_n \vdash B_1, B_2, \ldots, B_m$, where formulae A_i's and B_i's are called the *antecedents* and the *consequents*, respectively.

The inference rules transform sequents, possibly introducing subgoals, generating a proof tree. A proof terminates successfully when all branches terminate with a sequent where either any formula occurs both as an antecedent and as a consequent, or any antecedent is false, or any consequent is true.

The PVS theorem prover offers inference rules that directly implement the basic rules of the sequent calculus, or combine them into powerful *strategies* that may often prove goals with a single prover command, such as *assert* or *grind*, that apply several substitutions and simplifications in one step.

The PVS language is purely declarative, but the PVSio extension [18] can compute the value of ground (i.e., fully instantiated) applications of a function. The PVSio ground evaluator, included in the PVS environment, can then be used as an interpreter for functions declared in a PVS theory. This feature makes it possible to use the PVS environment as a prototyping and co-simulation tool.

4 Case Study: a Two-Wheeled Vehicle

Let us consider an abstract representation of a terrestrial vehicle with a pair of wheels connected through an axle, moving on a flat surface. This representation abstracts away the workings of all subsystems, such as propulsion and steering mechanisms. In a rectangular Cartesian frame, its configuration can be defined by three state variables: the coordinates x and y of the axle's midpoint, and the orientation ψ (*yaw* angle) of the vehicle's instantaneous direction with the x axis (Fig. 1).

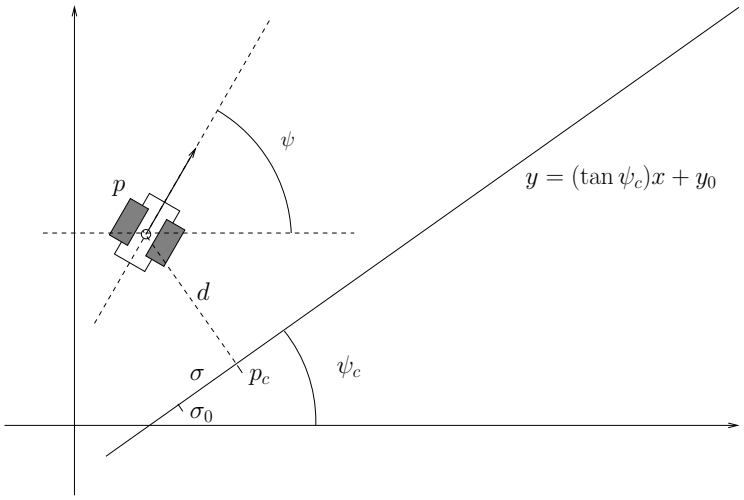


Fig. 1. Representation of the case study.

Let us further assume that the vehicle's linear speed v is a constant V. The only controlled variable is then the yaw angle, which must satisfy the relation $\dot{\psi} = \omega$, where ω is the rotational speed around the vertical axis of the vehicle imposed by the controller. The kinematics of the vehicle in the Cartesian reference frame are then given by the following system:

$$\begin{cases} \dot{x} = \mathrm{V}\cos\psi \\ \dot{y} = \mathrm{V}\sin\psi \\ \dot{\psi} = \omega \end{cases}$$

In this case study, the objective is to prove that a given control law is sufficient to lead the vehicle to move along an arbitrarily assigned target straight line. It is also required that the vehicle approaches the line smoothly, without oscillations.

Let the generic target line c be represented as

$$y = (\tan\psi_c)x + y_0 \ . \tag{1}$$

If $p = (x, y)$ is the vehicle's position and p_c is the orthogonal projection of p on c, then $d =\mid p - p_c \mid$ and $\theta = \psi - \psi_c$ are, respectively, the distance from the vehicle to the target line and the difference between the vehicle's direction and the direction of the target line. It is convenient to adopt a mobile reference frame with p_c as its origin and σ and d as the spatial coordinates, where σ is the distance of p_c from a reference point σ_0 on c. In this new frame, we have

$$\begin{cases} \dot{\sigma} = \mathrm{V} \cos\theta \\ \dot{d} = \mathrm{V} \sin\theta \\ \dot{\theta} = \dot{\psi} = \omega \end{cases} \tag{2}$$

The objective of the control law is then to have d and θ vanish asymptotically with time, and in particular we intend to verify that this can be accomplished with the control law

$$\omega = -dv \operatorname{sinc}\theta - \mathrm{k}\theta \tag{3}$$

where

$$\operatorname{sinc}\theta = \begin{cases} \frac{\sin\theta}{\theta} \text{ if } \theta \neq 0 \\ 1 \quad \text{otherwise} \end{cases}$$

and 'k' is a parameter of the control law.

5 Verification

In this section, the PVS interactive theorem prover is used to verify that the movement along the x-axis is locally asymptotically stable, i.e., that the trajectory of a vehicle, described by the kinematics of 2 and controlled by the law of 3, approaches the target line 1 as time approaches infinity, if the initial movement is contained within a "sufficiently small" neighborhood of the desired trajectory. Also, a condition on the control parameter k is found, guaranteeing that the movement is free of oscillation. The proof is very simple, and follows the common practice in control theory: The system is linearized at the desired state, the system's Jacobian is formulated, and the asymptotic stability of the state (without oscillations) is verified by showing that the eigenvalues are real and negative.

With the control law 3, the kinematics are given by this system of generating functions:

$$\begin{cases} \dot{\sigma} = f_\sigma(\sigma, d, \theta) & = \mathrm{V} \cos\theta \\ \dot{d} = f_d(\sigma, d, \theta) & = \mathrm{V} \sin\theta \\ \dot{\theta} = f_\theta(\sigma, d, \theta) & = -d\mathrm{V} \operatorname{sinc}\theta - \mathrm{k}\theta \ , \end{cases}$$

whose partial derivatives are

$$\begin{array}{lll} \frac{\partial f_\sigma}{\partial \sigma} = 0 & \frac{\partial f_\sigma}{\partial d} = 0 & \frac{\partial f_\sigma}{\partial \theta} = -V \sin\theta \\ \frac{\partial f_d}{\partial \sigma} = 0 & \frac{\partial f_d}{\partial d} = 0 & \frac{\partial f_d}{\partial \theta} = V \cos\theta \\ \frac{\partial f_\theta}{\partial \sigma} = 0 & \frac{\partial f_\theta}{\partial d} = -V \operatorname{sinc}\theta & \frac{\partial f_\theta}{\partial \theta} = -V d(\frac{\theta\cos\theta - \sin\theta}{\theta^2}) - k \ . \end{array}$$

The partial derivatives are expressed in the *intrinsic_kinematics* theory:

```
intrinsic_kinematics: THEORY BEGIN
IMPORTING sinc_th

t: VAR nnreal          % time
sigma(t): real         % sigma coordinate
d(t): real             % d coordinate
V: posreal             % linear velocity
k: posreal
theta(t): real         % angle between velocity and target line
omega(t): real         % turning speed

% partial derivatives of the generating functions
dfsigma_dsigma(sigma, d, theta: [nnreal -> real], t: real):
                                                        real = 0
dfsigma_dd(sigma, d, theta: [nnreal -> real], t: real):  real = 0
dfsigma_dtheta(sigma, d, theta: [nnreal -> real], t: real):
                                           real = -V*sin(theta(t))
dfd_dsigma(sigma, d, theta: [nnreal -> real], t: real):  real = 0
dfd_dd(sigma, d, theta: [nnreal -> real], t: real):      real = 0
dfd_dtheta(sigma, d, theta: [nnreal -> real], t: real):
                                            real = V*cos(theta(t))
dftheta_dsigma(sigma, d, theta: [nnreal -> real], t: real):
                                                        real = 0
dftheta_dd(sigma, d, theta: [nnreal -> real], t: real):
                                          real = -V*sinc(theta(t))
dftheta_dtheta(sigma, d, theta: [nnreal -> real], t: real):
                                                          real =
-V*d(t)*(cos(theta(t))/theta(t) - sin(theta(t))/(theta(t))^2) - k
END intrinsic_kinematics
```

In the above theory, the *importing* clause makes the *sinc_th* theory available, containing the definition of the 'sinc' function with an axiom defining its value at the origin. The initial declarations introduce t as the independent variable over non-negative reals (`nnreal`), σ, d, θ, and ω as real functions of t, and V and k as positive real (`posreal`) constants. The generating functions are then defined with four arguments: the three functions of time (of type [*nnreal* → *real*]) σ, d, and θ, and time t itself.

Another theory, *linearized_intrinsic*, defines the Jacobian that linearizes the system around the target configuration $d = 0$, $\theta = 0$:

$$J = \begin{bmatrix} \frac{\partial f_\sigma}{\partial \sigma} & \frac{\partial f_\sigma}{\partial d} & \frac{\partial f_\sigma}{\partial \theta} \\ \frac{\partial f_d}{\partial \sigma} & \frac{\partial f_d}{\partial d} & \frac{\partial f_d}{\partial \theta} \\ \frac{\partial f_\theta}{\partial \sigma} & \frac{\partial f_\theta}{\partial d} & \frac{\partial f_\theta}{\partial \theta} \end{bmatrix} = \begin{bmatrix} 0 & 0 & -V \sin\theta \\ 0 & 0 & V \cos\theta \\ 0 & -V \operatorname{sinc}\theta & -Vd(\frac{\theta\cos\theta - \sin\theta}{\theta^2}) - k \end{bmatrix}$$

In PVS, a matrix like this can be represented as a function of two natural numbers i and j, of the three state variables, and of time. The numbers i and j are the row and column indices of the matrix, which select one of the nine partial derivatives, which is then evaluated for the triple (σ, d, θ) and for the value t of time:

```
linearized_intrinsic: THEORY BEGIN
IMPORTING intrinsic_kinematics

% Jacobian
J(i, j: {n: posnat | n <= 3}, sigma, d, theta: [nnreal -> real], t)
                                                        : real =
    let idx = 3*(i - 1) + (j - 1)
    in cond
        idx = 0 -> dfsigma_dsigma(sigma, d, theta, t),
        idx = 1 -> dfsigma_dd(sigma, d, theta, t),
        ...
        idx = 7 -> dftheta_dd(sigma, d, theta, t),
        idx = 8 -> dftheta_dtheta(sigma, d, theta, t)
       endcond
```

In the above code, indices i and j are used to compute index *idx*, whose value is used in the *cond* clause to select one the nine elements of the Jacobian.

At the target configuration, the Jacobian reduces to

$$J_0 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & V \\ 0 & -V & -k \end{bmatrix}$$

as stated in the *linearized_intrinsic* theory by the *J_0* predicate:

```
% Jacobian at desired configuration
J_0(sigma, d, theta: [nnreal -> real], t) : bool =
    J(1, 1, sigma, d, theta, t) = 0 and
    ...
    J(2, 3, sigma, d, theta, t) = V and
    J(3, 1, sigma, d, theta, t) = 0 and
    J(3, 2, sigma, d, theta, t) = -V and
    J(3, 3, sigma, d, theta, t) = -k
```

The characteristic polynomial is

$$P(\lambda) = -\lambda^3 - k\lambda^2 - V^2\lambda$$

whose eigenvalues are

$$\lambda_1 = -\frac{\sqrt{k^2-4V^2}+k}{2} \qquad \lambda_2 = \frac{\sqrt{k^2-4V^2}-k}{2} \qquad \lambda_3 = 0\ .$$

Accordingly, the theory has these declarations:

```
% characteristic polynomial of J_0
char_J(lam: real) : real = -lam^3 - k*lam^2 - (V^2)*lam

% eigenvalues
lam_1: real = - (sqrt(k^2 - 4*V^2) + k)/2
lam_2: real =   (sqrt(k^2 - 4*V^2) - k)/2
lam_3: real =   0
```

Note that in this theory the eigenvalues are *declared* to be reals, due to the requirement that the vehicle's approach to the target line be free of oscillations. The PVS theorem prover keeps record of each variable's type and uses this information to generate automatically *type-check conditions* that must be discharged to complete a proof.

The above listings are the axiomatic part of the theory, i.e., the declarations needed to formalize the problem at hand. The verification part involves writing lemmas to be proved, stating the desired properties. In this case, it must be proved that the three values proposed as eigenvalues are indeed roots of the characteristic polynomial, and that they are real and nonpositive.

The correctness of J_0 is expressed by this lemma:

```
J_0_lem: LEMMA
    d(t) = 0 and theta(t) = 0 implies J_0(sigma, d, theta, t)
```

which is proved introducing two simple lemmas on the values of the sine and cosine functions at zero and applying the *assert* rule.

From the form of the candidate eigenvalues λ_1 and λ_2, it is clear that k and V must satisfy $k \geq 2V$ for the eigenvalues to be real. We may note that if this constraint is overlooked (as might happen in a more complex case), interactive theorem proving can help discover it. In fact, let us try to prove the following:

```
eigenvals: LEMMA    % FIRST ATTEMPT
    char_J(lam_1) = 0 and char_J(lam_2) = 0 and char_J(lam_3) = 0
```

The proof does not succeed, but the failure shows clearly what is missing, since several steps in the attempted proof generate this unsolvable goal:

```
k*k >= 4*(V*V)
```

i.e., the condition on k and V required to have real-valued eigenvalues, which can then be introduced in the *eigenvals* lemma:

```
eigenvals: LEMMA k > 2*V implies
    char_J(lam_1) = 0 and char_J(lam_2) = 0 and char_J(lam_3) = 0
```

The above lemma is used as a step to prove the conclusive lemma on local stability (labeled as a theorem just to set it aside from the preliminary steps):

```
local_stability: THEOREM k > 2*V implies
    char_J(lam_1) = 0 and char_J(lam_2) = 0 and char_J(lam_3) = 0
    and lam_1 < 0 and lam_2 < 0
```

The proof of both lemmas is straightforward, relying mostly on basic sequent transformations, expansion of definitions, and algebraic manipulations. The latter play a major role in the proof, and are carried out with dedicated rules from the *manip* package [7], simple lemmas from the prelude or the NASALIB libraries, or *ad hoc* lemmas that are proved with the *grind* rule.

6 Co-Simulation

In order to co-simulate the vehicle system, the control law has been expressed as a PVS theory executed by PVSio and the kinematics have been modeled and simulated with Simulink. The two simulations are coordinated by a module embedded in the Simulink model as an S-function, i.e., a user-defined block written in Matlab or, in this case, in C. More precisely, the Simulink model is composed of two subsystems: the vehicle kinematics and an S-function block that spawns the PVSio process executing the control law theory and then manages the communication between PVSio and Simulink.

6.1 Simulink Model of the Plant

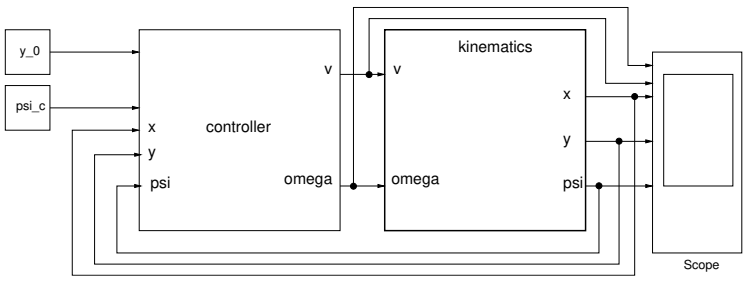


Fig. 2. Co-simulation model.

Figure 2 shows the complete Simulink model, where the *controller* block produces the ω and v inputs to the *kinematics* block. The outputs of the latter

are fed back to the controller, which also takes as input the parameters ψ_c and y_0 defining the target line.

The co-simulation is driven by the Simulink engine. At the beginning, the controller opens two Unix pipes for bidirectional communication with PVSio and spawns a PVSio process. At each subsequent simulation step, the controller block submits a request to evaluate a *step* function (explained below) for the current state. The interpreter's reply is parsed to extract the computed values of ω and v, which are returned to the Simulink engine.

The kinematics of the vehicle are modeled in Simulink as shown in Fig. 3. The turning speed ω is integrated to obtain the yaw angle ψ, which is fed to the blocks computing sine and cosine, their outputs are multiplied by the linear speed v, and the result is integrated to obtain the coordinates x and y.

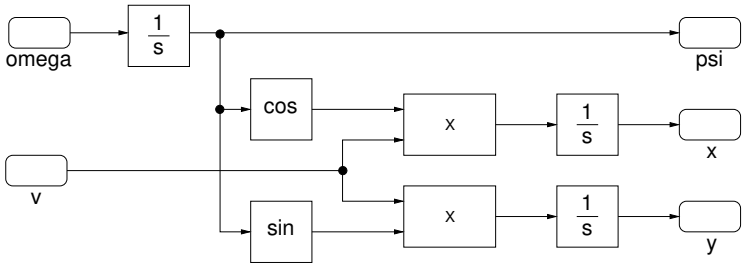


Fig. 3. Kinematics model in Simulink.

6.2 PVS Model of the Controller

In the previous sections, the partial derivatives of the generating functions have been expressed directly in the PVS notation and used for verification. In order to *simulate* the behavior of the controller, i.e., to produce a sequence of values for ω corresponding to discrete instants, we must represent the controller as a transition system whose state changes at each step according to the specified control law. This is done by defining a *state* data structure containing the instantaneous values of all the variables, and a *step* function that updates the state.

At each simulation step, the *controller* Simulink block reads the values of x, y, and ψ from the *kinematics* block, composes a *state* record containing this information along with other parameters, and encodes the record into an application expression of the *step* function. The PVSio interpreter evaluates the new value of ω according to the control law, and inserts it in the updated state, which is returned to the *controller* block on the Simulink side.

The controller is defined by the *controller* theory:

```
controller : THEORY BEGIN
IMPORTING stdmath
```

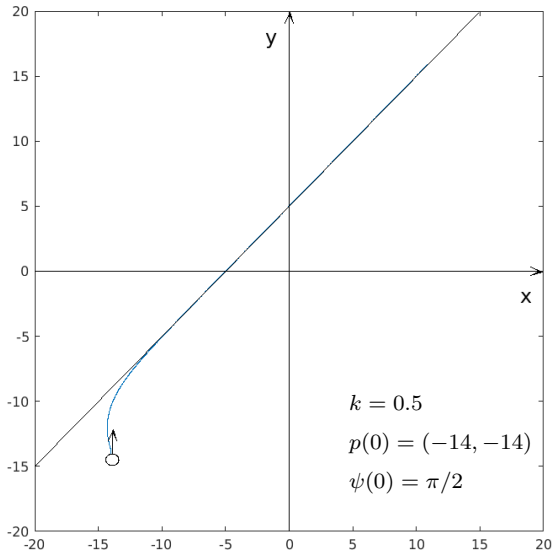


Fig. 4. Example simulation, $k = 0.5$.

```
State : TYPE = [#
    y: real, x: real, psi: real,      % inputs
    y_0: real, psi_c: real,           % target line
    k: real, v: real,                 % parameters
    w: real #]                        % output

sinc(angle: real) : real =
    IF (angle = 0) THEN 1.0 ELSE SIN(angle)/angle ENDIF
```

This theory uses the *stdmath* prelude theory, providing executable definitions of various functions, such as *SIN* and *COS*, to be used for simulation or prototyping purposes. These functions are logically equivalent to the corresponding functions, such as *sin* and *cos*, axiomatically defined in the NASALIB theories for verification purposes.

The *state* record type contains the vehicle coordinates and yaw angle coming from the Simulink side, the parameters of the target line, the control law parameters, and the turning speed to be sent to the Simulink side. The *step* function uses the input and parameter fields of the *state* record to compute the new value of ω and replace the previous one in a new copy of the record:

```
step(s:State): State = s WITH [w := -(y(s)*COS(psi_c(s))
    - x(s)*SIN(psi_c(s)) - y_0(s)*COS(psi_c(s)))
        *v(s)*(sinc(psi(s)-psi_c(s))) - k(s)*(psi(s) - psi_c(s)) ]
```

In the above code, expressions like *y(s)* or *psi_c(s)* denote the values of the fields y or *psi_c* of the state record s. The expression s *WITH* [$w := \ldots$] denotes a

copy of s, where the value of field w is replaced by a new value. Please note that, in spite of its assignment-like appearance, this expression is purely declarative: It is the PVSio ground evaluator that turns this declaration into executable code. The function applications produced by the Simulink *controller* block are similar to the following:

```
step((#y:=1,x:=0,psi:=0,y_0:=0,psi_c:=0,k:=0.5,v:=0.1,w:=0#));
```

Such expressions are generated by the S-function in the Simulink model and sent to the PVSio interpreter, which returns a string with a similar syntax, sent back to Simulink and parsed by the S-function.

It can be seen that the theory defining the controller is extremely simple. The relevant fact is that the controller theory used for co-simulation relies on the same control law as the one used for verification, without any need of translating it from one language (PVS) to another (Simulink), and therefore without any need of checking the equivalence between two forms of the same law.

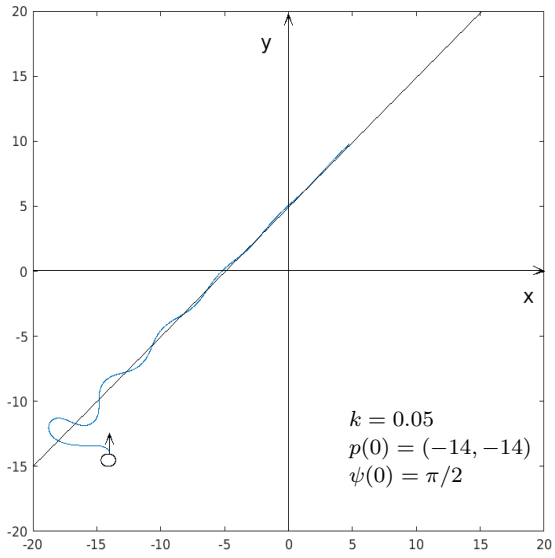
6.3 Examples of Co-Simulation

The co-simulation model has been exercised by varying the parameters of the target line and of the control law, and the initial position and orientation of the vehicle. For example, Fig. 4 shows the path of the vehicle starting at $(-14, -14)$ with $\psi(0) = \pi/2$ and approaching a straight line with $\psi_c = \pi/4$ and $y_0 = 5$. The control law parameters are $k = 0.5$ and $v = 0.2$. If parameter k is reduced to 0.05, the resulting trajectory has an oscillating transient, as shown in Fig. 5(a). In Fig. 5(b), the starting point is $(-10, 5)$ with $\psi(0) = 0$.

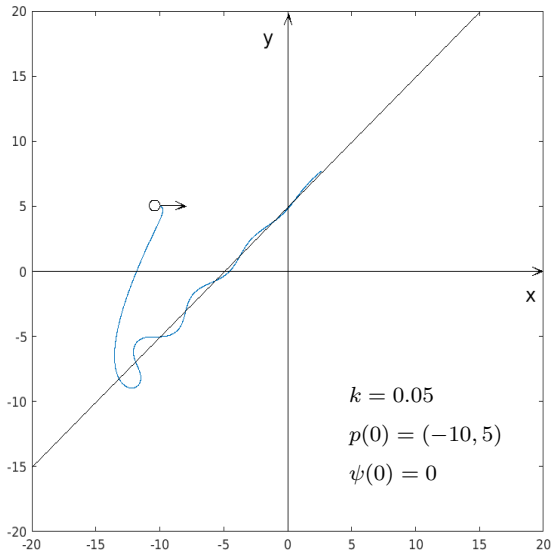
7 Conclusions

This paper proposes an approach to CPS development integrating co-simulation and formal verification, using a simple case study to provide a concrete example. The key concept is that both (co-)simulation and formal verification are necessary in this type of systems, and that both techniques can take advantage of environments where the same formalism is used to produce declarative models that can be both verified and executed. In this paper, it has also been shown how a PVS model can be co-simulated together with heterogeneous models.

In the case study reported in this paper, the higher-order language of the Prototype Verification System was used to model a simple autonomous vehicle and to specify its required behavior. The model of the vehicle consists in two parts: its kinematics and its control law. The theory defining the vehicle and its requirements has been used to verify that the control law complies with the requirements, provided that its parameters satisfy a relation needed in the proof. Concurrently, the control law has been used to build an executable model of the controller, which has been co-simulated with a Simulink model of the plant, i.e., the vehicle's kinematics, thus providing a validation of the verification results.



(a)



(b)

Fig. 5. Example simulations, $k = 0.05$.

For example, simulations have shown the system's behavior when the verification's assumptions are violated. This procedure has two useful consequences: (i) Using exactly the same controller model for verification and simulation avoids the effort both of producing two models of the same controller and of proving

their equivalence; and (ii) having different plant models for verification and simulation makes it possible to cross-check the two models. This case study has convinced the authors that interactive theorem proving can be used effectively and efficiently in the development of CPSs. Another relevant aspect of the case study is the co-simulation framework. In this case, integration of PVSio and Simulink has been achieved quite simply by embedding the PVSio interpreter in the Simulink model, using the S-function feature and the operating system primitives for process management and communication. More general approaches can be used, such as, e.g., the Functional Mockup Interface standard [5] used in the INTO-CPS tool [14], or the PVSio-web framework [21].

References

1. Bernardeschi, C., Cassano, L., Domenici, A., Sterpone, L.: ASSESS: A Simulator of Soft Errors in the Configuration Memory of SRAM-Based FPGAs. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 33(9), 1342–1355 (Sept 2014)
2. Bernardeschi, C., Cassano, L., Cimino, M.G., Domenici, A.: GABES: a Genetic Algorithm Based Environment for SEU Testing in SRAM-FPGAs. Journal of Systems Architecture 59(10, Part D), 1383–1254 (2013)
3. Bernardeschi, C., Domenici, A.: Verifying safety properties of a nonlinear control by interactive theorem proving with the Prototype Verification System. Information Processing Letters 116(6), 409–415 (2016)
4. Bernardeschi, C., Domenici, A., Masci, P.: A PVS-Simulink Integrated Environment for Model-Based Analysis of Cyber-Physical Systems. IEEE Transactions on Software Engineering PP(99), 1–1 (2017)
5. Blochwitz, T., Otter, M., Arnold, M., Bausch, C., Clauß, C., Elmqvist, H., Junghanns, A., Mauß, J., Monteiro, M., Neidhold, T., Neumerkel, D., Olsson, H., Peetz, J.V., Wolf, S.: The Functional Mockup Interface for Tool independent Exchange of Simulation Models. In: Proc. of the 8th International Modelica Conference. pp. 105–114. Linköping University Electronic Press (2011)
6. Carreño, V., Muñoz, C.: Aircraft trajectory modeling and alerting algorithm verification. In: Aagaard, M., Harrison, J. (eds.) Theorem Proving in Higher Order Logics, Lecture Notes in Computer Science, vol. 1869, pp. 90–105. Springer Berlin Heidelberg (2000)
7. Di Vito, B.: Manip User's Guide, Version 1.3, http://shemesh.larc.nasa.gov/people/bld/ftp/manip-guide-1.3.pdf, retrieved 8/18/2015
8. Dutertre, B.: Elements of mathematical analysis in PVS. In: Goos, G., Hartmanis, J., van Leeuwen, J., von Wright, J., Grundy, J., Harrison, J. (eds.) Theorem Proving in Higher Order Logics, Lecture Notes in Computer Science, vol. 1125, pp. 141–156. Springer Berlin Heidelberg (1996)
9. Fitzgerald, J.S., Larsen, P.G., Verhoef, M.: Vienna Development Method. John Wiley & Sons, Inc. (2007)
10. Fulton, N., Mitsch, S., Quesel, J.D., Völp, M., Platzer, A.: KeYmaera X: An axiomatic tactical theorem prover for hybrid systems. In: International Conference on Automated Deduction. pp. 527–538. Springer (2015)
11. Gomes, C., Thule, C., Broman, D., Larsen, P.G., Vangheluwe, H.: Co-simulation: State of the art. ACM Computing Surveys (2017, to appear)

12. Gottliebsen, H.: Transcendental Functions and Continuity Checking in PVS. In: Aagaard, M., Harrison, J. (eds.) Theorem Proving in Higher Order Logics, Lecture Notes in Computer Science, vol. 1869, pp. 197–214. Springer Berlin Heidelberg (2000)
13. Karnopp, D., Rosenberg, R.: Analysis and simulation of multiport systems; the bond graph approach to physical system dynamics. M.I.T. Press, Cambridge, MA, USA (1968)
14. Larsen, P.G., Fitzgerald, J., Woodcock, J., Fritzson, P., Brauer, J., Kleijn, C., Lecomte, T., Pfeil, M., Green, O., Basagiannis, S., Sadovykh, A.: Integrated tool chain for model-based design of Cyber-Physical Systems: The INTO-CPS project. In: 2016 2nd International Workshop on Modelling, Analysis, and Control of Complex CPS (CPS Data). pp. 1–6 (April 2016)
15. Larsen, P., Gamble, C., Pierce, K., Ribeiro, A., Lausdahl, K.: Support for Co-modelling and Co-simulation: The Crescendo Tool, pp. 97–114. Springer (2014)
16. Lee, E.A., Sangiovanni-Vincentelli, A.: A framework for comparing models of computation. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 17(12), 1217–1229 (Dec 1998)
17. Masci, P., Oladimeji, P., Zhang, Y., Jones, P., Curzon, P., Thimbleby, H.: PVSio-web 2.0: Joining PVS to HCI. In: Kroening, D., Păsăreanu, S.C. (eds.) Computer Aided Verification: 27th International Conference, CAV 2015, Proceedings, Part I. pp. 470–478. Springer International Publishing (2015), tool available at http://www.pvsioweb.org
18. Muñoz, C.: Rapid prototyping in PVS. Tech. Rep. NIA 2003-03, NASA/CR-2003-212418, National Institute of Aerospace, Hampton, VA, USA (2003)
19. Muñoz, C., Narkawicz, A., Hagen, G., Upchurch, J., Dutle, A., Consiglio, M.: DAIDALUS: Detect and Avoid Alerting Logic for Unmanned Systems. In: Proceedings of the 34th Digital Avionics Systems Conference (DASC 2015). Prague, Czech Republic (September 2015)
20. Niaki, S.H.A., Sander, I.: Co-simulation of embedded systems in a heterogeneous MoC-based modeling framework. In: 2011 6th IEEE International Symposium on Industrial and Embedded Systems. pp. 238–247 (June 2011)
21. Oladimeji, P., Masci, P., Curzon, P., Thimbleby, H.: PVSio-web: a tool for rapid prototyping device user interfaces in PVS. In: FMIS2013, 5th International Workshop on Formal Methods for Interactive Systems, London, UK, June 24, 2013 (2013)
22. Owre, S., Rajan, S., Rushby, J., Shankar, N., Srivas, M.: PVS: combining specification, proof checking, and model checking. In: Alur, R., Henzinger, T. (eds.) Computer-Aided Verification, CAV '96, pp. 411–414. No. 1102 in LNCS, Springer-Verlag (1996)
23. Owre, S., Rushby, J., Shankar, N., Srivas, M.: A tutorial on using PVS for hardware verification. In: Kumar, R., Kropf, T. (eds.) Theorem Provers in Circuit Design (TPCD '94), pp. 258–279. No. 901 in LNCS, Springer-Verlag (1997)
24. Platzer, A.: Logics of dynamical systems. In: Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science. pp. 13–24. LICS '12, IEEE Computer Society, Washington, DC, USA (2012)
25. Sanders, W.H., Meyer, J.F.: Stochastic activity networks: formal definitions and concepts. In: Lectures on formal methods and performance analysis: first EEF/Euro summer school on trends in computer science, pp. 315–343. Springer-Verlag New York, Inc., New York, NY, USA (2002)
26. Smullyan, R.M.: First-order logic. Dover publications, New York (1995)