https://doi.org/10.1093/comjnl/bxab161

Co-simulation and Formal Verification of Co-operative Drone Control With Logic-Based Specifications

Cinzia Bernardeschi[1,*], Andrea Domenici[1], Adriano Fagiolini[2] and Maurizio Palmieri[1]

[1]*Department of Information Engineering, University of Pisa, Pisa, Italy*
[2]*Department of Engineering, University of Palermo, Palermo, Italy*
[*]*Corresponding author: cinzia.bernardeschi@unipi.it*

Unmanned aerial vehicle (UAV) co-operative systems are complex cyber-physical systems that integrate a high-level control algorithm with pre-existing closed implementations of lower-level vehicle kinematics. In model-driven development, simulation is one of the techniques that are usually applied, together with testing, in the analysis of system behaviours. This work proposes a method and tools to validate the design of UAV co-operative systems based on co-simulation and formal verification. The method uses the Prototype Verification System, an interactive theorem prover based on a higher-order logic language, and the Functional Mock-up Interface, a widely accepted standard for co-simulation. In this paper, results on the co-simulation and proofs of safety requirements of a representative co-ordination algorithm are shown and discussed in a scenario where quadcopters are deployed and perform space-coverage operations.

Keywords: formal methods; co-operative control; co-simulation; verification; theorem prover

Received 24 November 2020; Revised 7 July 2021; Editorial Decision 9 September 2021
Handling editor: Mark Ryan

1. INTRODUCTION

Unmanned aerial vehicles (UAVs), or *drones*, are increasingly being used in a wide spectrum of activities, ranging from entertainment to building and plant inspection, archaeology, forestry management, crop dusting and search-and-rescue operations [50]. Such a widespread and varied usage makes it necessary to provide guarantees of their behaviour through robust software validation and verification processes enabling errors to be detected and corrected during system development. This work uses a problem in co-operative UAV control to present an approach aimed at enhancing dependability through the integration of simulation and formal verification.

UAVs, and more generally cyber-physical systems (CPSs), obey both a continuous-time physical plant dynamics and a hybrid control dynamics having both a discrete-time (event driven) and a continuous-time component. In particular, a major distinction appears between the physical subsystems, or plant, and the control subsystem.

The physical aspects of a system are usually modelled with block-based graphical environments, e.g. Simulink[1] or Scicoslab[2] , that use blocks as graphical representations of mathematical operations.

Block diagrams are used to express the physical laws governing the various subsystems, which may exhibit different aspects, such as mechanical, electromagnetic or thermal ones.

The control part has a continuous-time and a discrete-time component. The continuous-time component is based on control theory, which deals with properties, such as stability, related to the time-continuous behaviour of systems, and relies on mathematical analysis, in particular on the theory of differential equations. A set of differential or differential-algebraic equations (or their Laplace representation) constitutes an abstract model that can be verified mathematically and is usually modelled with block-based languages.

1 http://www.mathworks.com/products/simulink
2 http://www.scicoslab.org

The discrete-time component arises from the intrinsically discrete operation of digital controllers and from the need of switching mode of operation on reception of controls or on environmental changes. Block-based environments may also model the discrete-time component (e.g. with the Stateflow toolkit in Simulink). However, digital controllers may be better modelled with other specific formalisms, such as hybrid automata [24].

Model-based development [49] relies mainly on simulation to assess compliance to system requirements. First, *software-in-the-loop simulation* is performed to execute the designed control algorithm within the simulated system.

Successively, the correctness of the whole system is usually validated with *hardware-in-the-loop simulation* [12], or with testing against the real plant.

In this process, a large, monolithic block-based model is simulated. Industrial-strength, general-purpose modelling and simulation environments provide extensive libraries of simulated components from different areas of physics and technology, but it is often the case that field specialists rely on specific tools and languages. In this case, model-based development can benefit from techniques of *co-simulation* [23] to integrate heterogeneous sub-systems, each modelled and simulated with specific tools.

Simulation-centred processes are essential in the validation of system design, but simulation and testing cannot be exhaustive, and the limitations of simulation are the more apparent if the increasing complexity of CPSs is taken into account. Formal verification should then take a central role in the assessment of safety requirements, but experimental and formal methods are orthogonal both from the conceptual and the technical standpoint, as they require quite different modelling languages and methods. This is likely to cause organizational problems in developments that try to use both approaches, including communication issues between the two teams and possible inconsistencies between the models used for simulation and for verification.

This work aims at integrating simulation and verification, relying on executable and verifiable formal specifications of the control part and on co-simulation to cope with the heterogeneity of the control and plant subsystems.

In particular, a method to support the analysis of UAV co-operative systems with this integrated approach is presented in this paper. The method is based on a higher-order logic language, and a single, logic-based specification is used both for verification and simulation of co-operative drone control. Using the same model for verification and simulation makes the process of cross-checking the results of both activities more reliable.

As a case study, a variant of a consensus protocol studied by Olfati-Saber *et al.* [41] has been considered. As discussed in Section 5, this variant is difficult to study along the lines proposed in that work. Therefore, the problem has been analysed with a higher-order logic specification of the algorithm. The algorithm has been specified and verified with the Prototype Verification System (PVS) [42] interactive theorem prover, which allowed us to prove convergence of the co-ordination protocol and collision avoidance under specific initial conditions.

The same PVS specification (with minor changes explained in Section 5) has been simulated using an interpreter for the PVS language. The whole system has then been co-simulated with the INTO-CPS framework, using Modelica and C to model the behaviour of the single drones. Besides affording the combined advantages of simulation and formal verification in the area of CPSs, this approach allows developers to independently replace both the physical model of drones and the co-ordination protocol as co-simulation components. So far this method has been applied to simple CPSs [17, 43], but it should be noted that it is applicable to a wide range of autonomous systems.

The main contribution of this paper is a development process for control systems of CPSs that integrates simulation and formal verification. A logic-based model of the control is used for verification and also co-simulated with models of the plant subsystems built with various simulation-oriented languages.

The paper is organized as follows: Section 2 presents some related work; Section 3 provides background on co-operative UAVs and on the consensus-problem, on logic-based formal specification and theorem proving and on co-simulation; Section 4 introduces and discusses the proposed approach; Section 5 introduces the case study; Section 6 reports co-simulation results of different scenarios; Section 7 describes how the logic model has been used to prove properties of the system; and Section 8 concludes the paper. An example of an interactive proof is shown in the Appendix.

2. RELATED WORK

Digital controllers may be modelled with formalisms specifically developed to deal with discrete-time behaviours. One class of such formalisms is based on state machines and in particular on hybrid automata [24], a conceptual model that lends itself to the integration of discrete- and continuous-time behaviours. Another class collects the logic-based methods, which use various forms of logic languages to model and analyse systems. These logic languages include temporal logics [34], normally used in conjunction with state-machine representations, and higher-order logics [32].

UAV simulation and application development rely on frameworks based on off-the-shelf tools, such as Gazebo [29] and Ardupilot[3] , which provide prototypes for the vehicle dynamics.

The work of Olfati-Saber *et al.* [41] is a useful starting point for the literature on coordination of autonomous vehicles and

[3] http://ardupilot.org

it introduces the consensus protocol that inspired our work. A survey of control problems was published by Ren *et al.* [47]. A review on communication architectures and routing protocols is in [13]. Among the many works on the coordination of autonomous vehicles or agents, we may also cite Fax and Murray [18] and Jadbabaie *et al.* [26]. A topic related to mobile autonomous agents is mobile sensing networks, addressed, e.g. by Cortés *et al.* [14] or Kar *et al.* [28].

A recent review on co-simulation has been published by Gomes *et al.* [23]. Blochwitz *et al.* [10] present the Functional Mock-up Interface (FMI), an emerging standard for co-simulation of CPS. The INTO-CPS framework, used in this work, is presented by Larsen *et al.* [30]. Another co-simulation framework is the HybridSim tool-chain, discussed by Wang and Baras [52], which transforms multi-domain models into a SysML [25] model. Jalali *et al.* [27] propose a framework for multisimulation based on a transaction-based approach for synchronization and analysis of dependencies among sub-models. Attarzadeh Niaki and Sander [2] address co-simulation of embedded systems extending the ForSyDe [48] framework. An extension to the FMI standard addressing the issue of time representation is discussed by Cremona *et al.* [15], and the problem of simulation stability is addressed by Gomes *et al.* [22].

Proposals to apply formal methods to CPSs follow many different approaches. For example, Dynamic Logic [11] is used with the KeyMaeraX [45] theorem prover, which has been integrated with the SPIRAL environment [46] as reported by Franchetti *et al.* [20]. The *Vienna Definition Method* (VDM) [19] family of languages and tools, in particular the *Crescendo* tool [31] have also been used extensively. Another important family of languages applied to CPSs is that of hybrid automata [24], and in particular timed automata [1] such as those supported by the UPPAAL environment [3].

In the present work, the PVS theorem prover [42] and the PVSio interpreter [39] have been used. PVS has been used in many application fields, including the formal verification of detect-and-avoid algorithms for Unmanned Aircraft Systems [36] and the validation of code for the same systems [37]. PVS has been used to assess properties of a simple non-linear CPS [6] and Newell *et al.* [38] used it to guarantee safety in a Nuclear Power Generating Station shutdown sub-system. PVS has been used in the field of medical devices and e-health [7, 35]. A tool based on PVSio for the simulation of user interfaces (PVSio-web) has been developed by Oladimeji *et al.* [40].

In previous work, the PVS/PVSio environment was used to simulate very simple single systems [17]. A first example of the integration of a PVS specification in a co-simulation was reported by Bernardeschi *et al.* [8], and the integration of PVS with FMI has been described in [43], where a simple robotic system, made up of a control part modelled in PVS and a continuous part modelled in 20-sim, has been co-simulated. In [9], the role of formal verification in model-driven development was discussed in reference to simple case studies, while in [5] design parameters for a complex control algorithm were studied using formal verification.

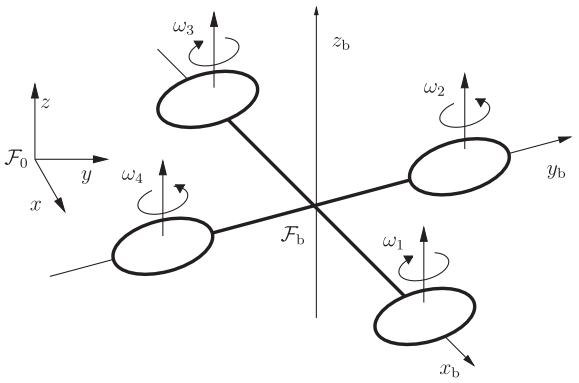

FIGURE 1. Schematic representation of a quadcopter.

3. BACKGROUND

This section briefly introduces notions used in the rest of the paper, concerning UAVs and UAV coordination, the PVS and the INTO-CPS platform for co-simulation.

3.1. Modelling co-operative UAVs

A quadrotor aircraft [33], or quadcopter, schematically consists, from the dynamical standpoint, in a cross-shaped frame supporting one rotor at each arm's end (Fig. 1). Each of the four rotors is a propeller driven by an electric motor, and each motor is controlled independently. The quadcopter's movements are determined by the resultant thrusts and torques of the rotors, which in turn depend on their angular speeds $\omega_1, \ldots \omega_4$.

The movement of a quadcopter can be described in terms of the displacement and velocity of its centre of mass with respect to a fixed inertial orthogonal reference frame $\mathcal{F}_0$ (i.e. the ground), and of its attitude, i.e. the orientation of its body with respect to the fixed reference frame. Velocity and attitude are related, since attitude determines the direction of thrust. For example, if the rotors lie on a horizontal plane and all turn at the same speed, the resultant thrust is in the vertical direction and the quadcopter can only move up and down, or hover at constant height when the thrust equals the weight. In order for the thrust (and thus velocity) to have a horizontal component, the quadcopter must tilt, changing the attitude. This is accomplished by driving the rotors at different speeds so that the different thrusts create torques on the quadcopter. Controlling a quadcopter therefore consists in setting the four rotor speeds to perform the desired manoeuvres.

To define the attitude, a moving orthogonal reference frame $\mathcal{F}_\mathrm{b}$ is introduced, with unit vectors $\hat{\imath}_\mathrm{b}$, $\hat{\jmath}_\mathrm{b}$ and $\hat{k}_\mathrm{b}$. The axes defined by $\hat{\imath}_\mathrm{b}$ and $\hat{\jmath}_\mathrm{b}$ coincide with the arms of the frame and

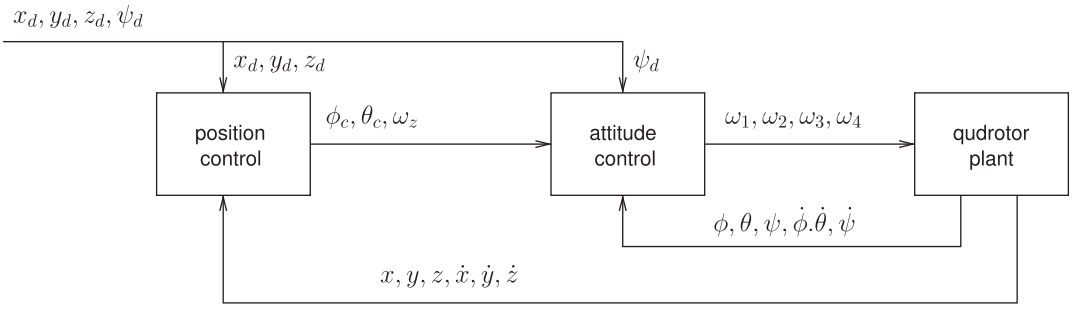

FIGURE 2. Cascaded position-attitude control.

have their origin at their crossing, which is also the centre of mass of the quadcopter. More precisely, the fixed reference frame $\mathcal{F}_0$ is defined by the fixed origin p_0 and axes x, y and z, with unit vectors $\hat{\imath}_0$, $\hat{\jmath}_0$ and $\hat{k}_0$. Axes x and y define the horizontal plane. Frame $\mathcal{F}_\mathrm{b}$ is defined by the origin $p_\mathrm{b} = (x, y, z)^T$ coincident with the quadcopter's centre of mass, and by axes x_b, y_b and z_b with unit vectors $\hat{\imath}_\mathrm{b}$, $\hat{\jmath}_\mathrm{b}$ and $\hat{k}_\mathrm{b}$.

The attitude can then be defined as the rotation that brings axes x_b, y_b and z_b to coincide with x, y and z, after having applied a translation that brings p_b to coincide with p_0. This rotation is the composition of three elementary rotations defined by the Euler angles ψ, ϕ and θ, also called *yaw*, *roll* and *pitch*, respectively. If $R(\psi, \phi, \theta)$ is the rotation matrix, any vector in $\mathcal{F}_\mathrm{b}$ can be expressed in $\mathcal{F}_0$ by multiplying it by $R(\psi, \phi, \theta)$.

With the above notations, the quadcopter's behaviour as a function of the rotor speeds can be described by two linearized models, one for position (equation 1) and one for attitude (equation 2). In the models, C_ϕ, C_θ, C_ψ and C_z are physical constants of the quadcopter related to mass, rotational inertia, geometry and rotor characteristics, g is the gravity acceleration and ψ_d is the desired yaw angle.

$$\begin{pmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{pmatrix} = \begin{pmatrix} g(\phi \sin\psi_d + \theta \cos\psi_d) \\ -g(\phi \cos\psi_d + \theta \sin\psi_d) \\ C_z(\omega_1 + \omega_2 + \omega_3 + \omega_4) \end{pmatrix} \tag{1}$$

$$\begin{pmatrix} \ddot{\phi} \\ \ddot{\theta} \\ \ddot{\psi} \end{pmatrix} = \begin{pmatrix} C_\phi(\omega_2 - \omega_4) \\ C_\theta(\omega_3 - \omega_1) \\ C_\psi(\omega_1 - \omega_2 + \omega_3 - \omega_4) \end{pmatrix}. \tag{2}$$

3.1.1. *Control models*

It is convenient to split the control in two cascaded modules, one for position and one for attitude (Fig. 2). The reference inputs of the position controller are the *desired* position (x_d, y_d, z_d) and yaw (ψ_d), while the respective actual values and speeds are the feedback inputs. The outputs are the *commanded* values of roll and pitch (ϕ_c and θ_c) and a value ω_z depending on the desired height above sea level (or altitude) z_d. These outputs are the reference inputs to the attitude control, which receives the actual Euler angles and their derivatives as feedbacks, producing the speed values for the rotors.

It may be shown [33] that a position controller can be defined by an equation of the following form, where $e_x = x - x_d$, $e_y = y - y_d$ and $e_z = z - z_d$ are the tracking errors; moreover, $\mathrm{s}_d = (\sin\psi_d)/g$ and $\mathrm{c}_d = (\cos\psi_d)/g$:

$$\begin{pmatrix} \phi_c \\ \theta_c \\ \omega_z \end{pmatrix} = \begin{pmatrix} -\mathrm{s}_d(2\lambda_P\dot{x} + \lambda_P^2 e_x) + \mathrm{c}_d(2\lambda_P\dot{y} + \lambda_P^2 e_y) \\ -\mathrm{c}_d(2\lambda_P\dot{x} + \lambda_P^2 e_x) - \mathrm{s}_d(2\lambda_P\dot{y} + \lambda_P^2 e_y) \\ -\frac{2\lambda_P}{C_z}\dot{z} + \frac{\lambda_P^2}{C_z} e_z) \end{pmatrix} \tag{3}$$

and an attitude controller can be defined by equations of the following form:

$$\begin{pmatrix} \omega_\phi \\ \omega_\theta \\ \omega_\psi \end{pmatrix} = \begin{pmatrix} -\frac{2\lambda_A}{C_\phi}\dot{\phi} - \frac{\lambda_A^2}{C_\phi}(\phi - \phi_c)) \\ -\frac{2\lambda_A}{C_\theta}\dot{\theta} - \frac{\lambda_A^2}{C_\theta}(\theta - \theta_c)) \\ -\frac{2\lambda_A}{C_\psi}\dot{\psi} - \frac{\lambda_A^2}{C_\psi}(\psi - \psi_d)) \end{pmatrix}$$

$$\begin{pmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \\ \omega_4 \end{pmatrix} = \frac{1}{4}\begin{pmatrix} \omega_z - 2\omega_\theta + \omega_\psi \\ \omega_z + 2\omega_\phi - \omega_\psi \\ \omega_z + 2\omega_\theta + \omega_\psi \\ \omega_z - 2\omega_\phi - \omega_\psi \end{pmatrix}. \tag{4}$$

In the equations above, λ_P and λ_A are gain parameters of the controller, which affect the speed of tracking error convergence.

3.1.2. *Co-operative UAVs*

The *consensus* problem is an important aspect of cooperation among autonomous agents, which can be described as the problem of reaching '*an agreement regarding a certain quantity of interest that depends on the state of all agents* [41].' This general concept applies to many different issues involving cooperative agents, among which the spatial distribution of a swarm of UAVs.

To formalize this particular problem, a graph $G = (V, E)$ is used, where the set V of vertices represents the set of UAVs and

the set E of edges represents the set of links among the UAVs that communicate and interact with each other, by exchanging information for the accomplishment of the task. For simplicity, we assume that the links are symmetric (i.e. the graph is undirected) and static (i.e. the topology does not change).

Each vertex i has a nonempty set $\mathbb{N}_i = \{j \in V \mid (i,j) \in E\}$ of *neighbours*, i.e. the set of agents with whom agent i interacts. The graph is characterized by three matrices: (i) the *adjacency* matrix A, whose elements or *weights* a_{ij} have a value equal to 1 if $(i,j) \in E$ or zero if not; (ii) the diagonal *degree* matrix D, whose elements d_{ii} are equal, respectively, to the number of links incident on vertex i; and (iii) the *graph Laplacian* L, defined as $D - A$, whose elements l_{ij} are such that

$$l_{ij} = \begin{cases} -1 & \text{if } j \in \mathbb{N}_i \\ |\mathbb{N}_i| & \text{if } j = i \\ 0 & \text{otherwise .} \end{cases}$$

Let then x_i be a state variable of agent i representing the quantity on which an agreement must be achieved, i.e. a global state $x = (x_1, \ldots, x_n)^T$ must be reached where $x_1 = x_2 = \ldots = x_n$. Olfati-Saber *et al.* [41] have shown that the law

$$\dot{x} = -\mathrm{L}x \tag{5}$$

is a distributed *consensus protocol*, i.e. a group of agents following this law will asymptotically approach an equilibrium state satisfying the consensus condition. Further, the consensus value will be the average of the initial values, for a connected graph. If the state variables represent, e.g. each agent's position, the above equation represents a *rendez-vous* protocol causing all agents to converge to the same position.

Equation (5) can be expressed in discrete time [41] by approximating $\dot{x}(k)$ according to Euler's discretization rule, i.e. $\dot{x}(k) \simeq (x(k+1) - x(k))/\epsilon$, where ϵ is the discretization step and k ranges over the nonnegative integers. The discrete-time form of (5) is then

$$x_i(k+1) = x_i(k) + \epsilon \sum_{j \in \mathbb{N}_i} a_{ij}(x_j(k) - x_i(k)) \ . \tag{6}$$

The discrete time collective dynamics of the set of agents can be rewritten in matrix form:

$$x(k+1) = \mathrm{P}x(k) \tag{7}$$

with $\mathrm{P} = I - \epsilon\mathrm{L}$ (I is the identity matrix). P is referred to as the *Perron* matrix [41].

Formation control. Equation (5) can be modified to adapt the consensus protocol to the case of formation control where the goal is to achieve a given spatial distribution. In this case, if x_i is the position of agent i and b_i is the vector sum of the distances from agent i to its neighbours, the consensus protocol becomes

$$\dot{x} = -\mathrm{L}x + b. \tag{8}$$

The protocol described by the above equation is called a *coverage* protocol in the following, as it is often used for area coverage applications, such as search-and-rescue or crop dusting. We may note that the input vector b is, by construction, orthogonal to the kernel of the Laplacian matrix, and thus it does not prevent the existence of a steady state.

As shown in [41], asymptotic convergence is guaranteed for systems of the above forms. Section 5 will refer to these results to discuss the case study presented in this paper, pointing out the differences that justify an alternative approach.

3.2. The PVS

The PVS [42] is an interactive theorem prover, enabling users to define theories and prove theorems within them. Theories are written in a typed higher-order language, where the user can define complex types and express properties of higher-order concepts, such as functions and sets. The theorem prover provides an extensive number of inference rules based on the sequent calculus [51], which the user can select and apply in different proof steps. The proof is not fully automatic but computer-assisted; the inference rules, however, are very powerful and experienced users may find proofs in a short time.

An additional feature of PVS is that it can also be used as a prototyping tool. This use is made possible by the PVSio extension. PVSio [39] is a ground evaluator that computes the value of ground (variable-free) expressions. The evaluator can also compute functions with side effects, such as producing outputs. It should be noted that the PVSio functions with side effects do not assign values to variables, and thus are logically equivalent to the normal (i.e. purely logical) functions of the PVS language, so that they do not interfere with theorem proving. The PVS theorem prover can be started in PVSio mode, where it accepts inputs in the form of ground function applications to evaluate. In this mode, the PVSio evaluator is used as an interpreter for a logic programming language.

3.2.1. *PVS language and sequent calculus*

The PVS language provides an extensive set of base types, including naturals, integers, reals and booleans, each defined by mathematical axioms in theories that comprise fundamental theorems. Therefore, these types represent the corresponding mathematical concepts, and not the finite and discrete approximations used by imperative programming languages. Various constructors are used to define complex types such as sets, tuples or records. In particular, *function types* are declared with type expressions of the form [*domain* → *codomain*], where *domain* and *codomain* can be any type, including function types. Functions with the Boolean codomain type are called *predicates*. A *formula* is an expression composed of variables,

constants, functions and operators, that evaluates to a Boolean value. Formulas assumed to be valid are labelled as *axioms*, whereas they are labeled as *lemmas* or *theorems* if they must be proved.

A *theory* is a collection of declarations and formulas, and a PVS specification consists in one or more theories. A theory may refer to other theories made accessible by *IMPORTING* declarations. The fundamental theories of the *prelude* library are imported implicitly, and additional libraries provide a large number of theories containing standard definitions and proved facts, e.g. about sets, sequences and graphs. A theory can also be defined in terms of parameters that are instantiated by importing theories. The basic constructs of the PVS language will be shown in examples throughout the paper. Some constructs and conventions, however, are introduced below: comments extend from a '`%`' character to the end of line. A *record* is a tuple whose elements are accessed through their respective *field* name, i.e. a record type is a shorthand for the Cartesian product of a number of sets. For example, given the declarations below,

```
polar: TYPE = [#  rho: real, theta: real #]
% record type
p: polar = (# rho := 1.0, theta := 2.0 #)
% record literal
```

the expressions *rho*(*p*) and *theta*(*p*) denote the modulus and argument of *p*. Equivalent notations are *p*'*rho* and *p*'*theta*.

Function declarations are in the form `foo(x: T1): T2`, where *foo* is the function name, *x* is a function argument of type *T1*, and *T2* is the function codomain type.

The PVS proof system is based on the *sequent calculus* [51]. A *sequent* is an expression of the form $A_1, A_2, \ldots, A_n \vdash B_1, B_2, \ldots, B_m$. The A_i's are called *antecedents* and the B_i's *consequents*. The '$\vdash$' symbol (the *turnstile*) is read as '*entails*' or '*yields*', meaning that the disjunction of the consequents can be deduced from the conjunction of the antecedents. Hence, in a proved sequent, when every A_i is true at least one of the B_i's must be true. Each antecedent or consequent is a formula, but not another sequent[4] .

The inference rules of the sequent calculus transform sequents. Some of the rules produce two or more new sequents from one sequent, so that the graph of intermediate subgoals linked by inference steps takes the form of a tree whose nodes are sequents and whose arcs are applications of inference rules. A proof terminates successfully when all branches terminate with a proved sequent, i.e. one where either any formula occurs both as an antecedent and as a consequent, or any antecedent is false, or any consequent is true.

[4] A sequent is an expression of the metalanguage of the deduction system, not a formula of the underlying logic language.

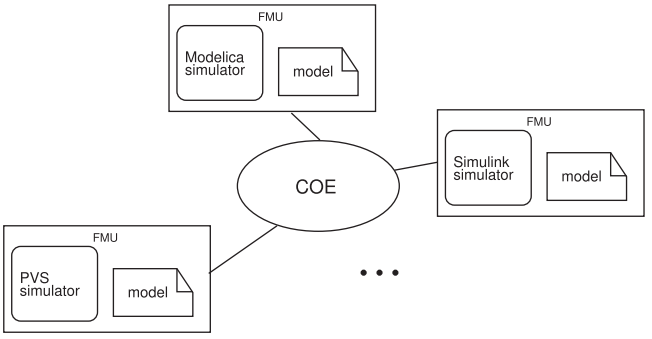

FIGURE 3. Co-simulation framework.

3.3. Co-simulation

Co-simulation (or *coupled simulation*) is the co-ordinated execution of two or more simulators, each running a distinct sub-model of an overall *multi-model*. This has some advantages over monolithic simulation of a complete system: In particular, it makes it possible to easily assemble a large simulation model out of pre-existing and possibly independently developed models. This is especially important for the simulation of a CPS, as it may be often built from physical components not originally conceived for the intended application of that CPS. Further, co-simulation gives developers the flexibility of choosing the fittest modelling formalism or the most convenient simulator for each sub-model, and, finally, the different simulators may run in parallel on networked machines, with a significant gain in performance.

An approach to co-simulation that is gaining acceptance is based on three concepts: (i) *functional mockup units* (FMU), i.e. software components packaging all that is needed to simulate a sub-model; (ii) the standard *FMI* defining the interface of the FMUs; and (iii) a *master algorithm* to co-ordinate the FMUs. The interface of an FMU includes, among others, *set* functions to set the values of the FMU's input variables, *get* functions to read the values of output variables, and the *doStep* function to request the execution of one simulation step.

In this work, the master algorithm developed by the INTO-CPS project [30] is used. INTO-CPS was a European project that developed an integrated tool-chain (Fig. 3) for model-based design of CPSs, based on the FMI standard. A core component of the tool-chain is the Co-simulation Orchestration Engine (COE), a validated FMI-compliant implementation of a master algorithm.

4. HIGHER-ORDER LOGIC FOR SPECIFICATION AND CO-SIMULATION

UAV specification and simulation must face the complexity of UAV systems: the intrinsic dynamical complexity of a single

drone, the complexity of interactions among members of a drone swarm and of interaction with the environment.

This paper proposes a process of performing co-simulation structured into four activity flows. In particular, three *modelling* flows (*PVS-based*, *equation-based* and *Modelica-based*) produce, possibly in parallel, three FMUs, each containing the respective models and tools. The PVS-based activity is a relevant and characterizing aspect of the proposed approach, as it makes it possible to use the same logic specification for simulation and formal verification. The other two modelling flows follow standard practices. In the equation-based activity, the model is C code implementing the controller equations and the FMU contains the corresponding executable code. In the Modelica-based activity, the translational and rotational dynamics are expressed in Modelica, and the OpenModelica environment produces the FMU.

The fourth activity is carried out by the *Master algorithm,* which coordinates the execution of the FMUs (three for each drone) and produces plots of simulation results.

The rest of this section introduces a general pattern for PVS models of drone swarms and their integration in a co-simulation architecture. The application of the complete process to a case study will be described in the next section.

4.1. A pattern for the specification of co-ordination protocols

In this section, we sketch a pattern of PVS theories for the modelling and simulation of a drone swarm. Later on (Section 5.3), actual theories for a specific case (Section 5) are shown.

Theory *system* (Listing 1) provides two basic definitions for the minimal information needed to identify a single drone, i.e., its identifier and its location in space, and then the definitions of record types to model the state of a single drone (*state*) and the collective state of a drone swarm (*systemState*). This theory is parametric in the number N of drones composing a swarm, and the identifiers range on the interval from 0 to $N-1$, represented by the PVS type *below(N)*.

The information defining the state of a single drone includes its identifier and location and possibly other items depending on the chosen protocol. Also, the swarm state will generally contain protocol-dependent items, but it can be assumed that the communication topology, the set of drone states, and a clock will be present. The topology is represented by field *g* whose type is defined in another theory (not shown). The set of drone states is represented by field *drones* whose value is a function of type *dmap* mapping each drone identifier to the respective state. The clock *timesteps* can be used as a counter of executed simulation steps.

Theory *system_execution* (Listing 2) is a pattern for simulation at swarm level. It defines a recursive function *kth_step* that computes the swarm state after k steps of simulation, starting from an initial state. Both the *initial* state and the *protocol* function representing a protocol step are left uninterpreted.

```
system[N: posnat]: THEORY
BEGIN

% state of a drone
drone_id: TYPE = below(N)
location: TYPE =
    [# x: real, y: real, z: real #]
state: TYPE =
    [# id: drone_id, self: location,
%       ... other fields #]

dmap: TYPE = [drone_id -> state]

% swarm of drones
systemState: TYPE =
    [# g: network_graph,
       drones: dmap,
       timesteps: nat #]
END system
```

LISTING 1. Definitions pattern for a drone swarm

```
system_execution[N: posnat]: THEORY
BEGIN IMPORTING system[N]

initial: systemState
protocol(s: systemState): systemState

kth_step(k: nat):
    RECURSIVE systemState =
    IF (k = 0) THEN initial
    ELSE protocol(kth_step(k - 1))
    ENDIF
MEASURE k
END system_execution
```

LISTING 2. Execution pattern for a drone swarm.

In the theory, *protocol* is a function that transforms the state of the swarm and depends on the identity of the single drone executing the protocol, thus allowing the execution of distributed protocols with possibly different behaviours of nodes.

Note that the *MEASURE* annotation in the theory enables the PVS type checker to control if recursion on k is well founded.

4.2. A Co-simulation architecture

The co-simulation architecture adopted in this experiment uses three specialized interacting FMUs for each drone (Fig. 4): a *plant* FMU for the drone dynamics, a *control* one to control attitude and velocity and a *coordination* one implementing the consensus protocol. The latter FMU receives the positions x_i of the neighbouring drones and computes the next desired position x_d, which is fed to the control FMU and transmitted to the neighbours' co-ordination FMUs. The control FMU reads the feedback from the plant, consisting in the current position x and attitude (ϕ, θ, ψ) and the respective time derivatives. The output is the quadruple of rotor speeds $(\omega_1, \omega_2, \omega_3, \omega_4)$, fed to the plant FMU.

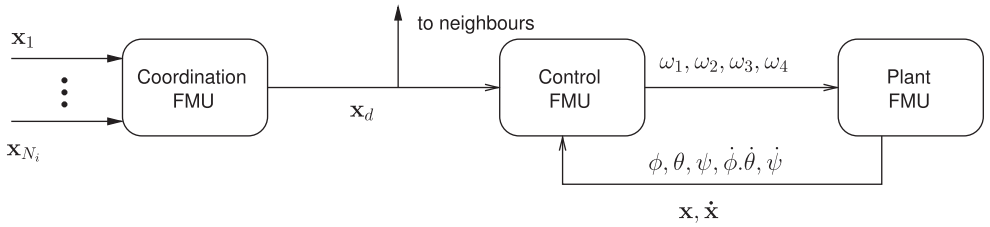

FIGURE 4. Logical connections between FMUs of a drone.

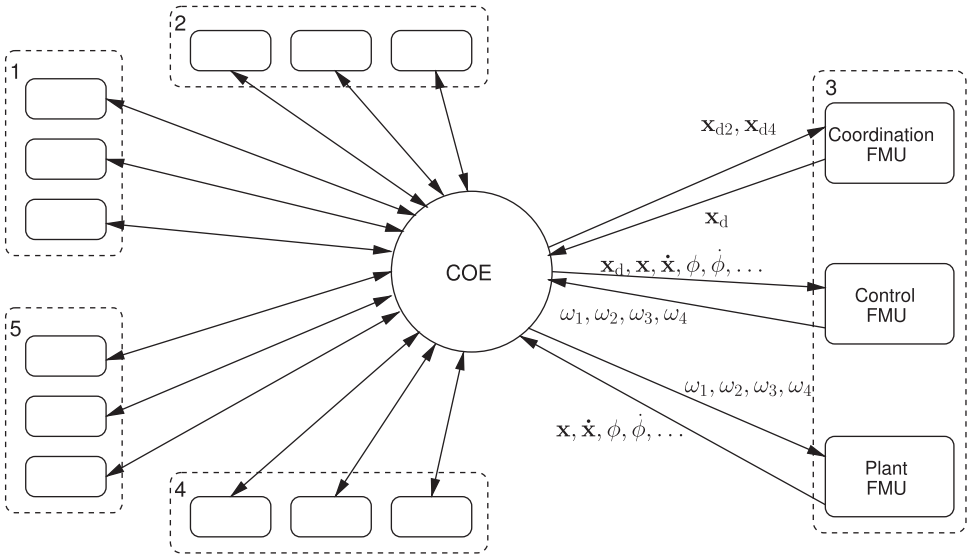

FIGURE 5. Co-simulation schema of a drone swarm.

The architecture of a co-simulation in a scenario of five quadcopters is shown in Fig. 5, where the variables without an index refer to drone 3. All the FMUs are connected through the INTO-CPS co-simulation engine.

4.3. Synchronization

In the simulation of co-operative UAV systems, drones communicate with each other through the Coordination FMU to execute the co-ordination protocol. Moreover, the Coordination FMU of a drone periodically sends the new target position to the Controller FMU of the drone. Then, given a target position, the two other FMUs of the drone (Controller FMU and Plant FMU) communicate with each other to steer the drone to the target position.

Generally, the rate of communication in the two cases above is different. We distinguish between the following:

- the time discretization interval ϵ at which the target position of each drone is updated in the coordination algorithm, and
- a smaller step τ that is the numeric integration step used by the drone dynamics simulator for communications with the low level controller of the drone.

In terms of the co-simulation architecture, ϵ is the interval at which the coordination FMUs of each drone compute the new target position and send it to the immediate neighbours and to the Control FMU of the drone and τ is the interval at which the simulator for the drone dynamics (part of the plant FMU) and the simulator of the attitude/position controller (part of the control FMU) exchange data.

5. APPLICATION TO A SPECIFIC CASE

Theories for a specific protocol can be developed after the pattern of the theories introduced in Section 4.

As an application example, we consider a special case of formation control, namely the uniform placement of a number of drones along a straight line segment. This placement could be, for example, the starting position in a search-and-rescue operation where the drones must explore parallel strips of terrain or in agricultural settings such as sowing operations.

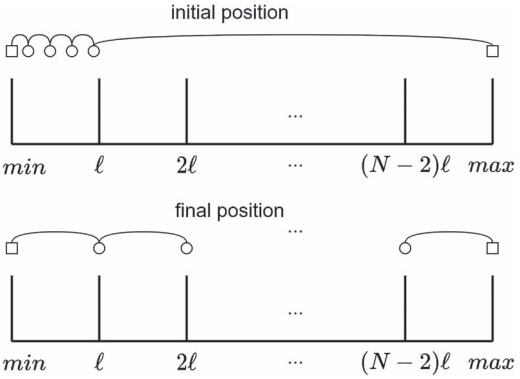

FIGURE 6. Initial and final positions of the drones.

With reference to Fig. 6, the following assumptions are made, with N, i and k ranging over the natural numbers:

- There are N drones, with $N \geq 3$, numbered consecutively from 0 to $N-1$;
- the segment is described by the interval $[min, max]$;
- the segment is divided in $N-1$ subsegments of equal length $\ell = (max - min)/(N-1)$;
- in the initial position,
  - drones 0 and $N-1$ (represented as squares in the picture) are placed at *min* and *max*, respectively;
  - the other drones (represented as circles) are aligned between *min* and ℓ, in the order of the respective indices;
- each drone can reliably communicate with its preceding and following immediate adjacent drone, with drones 0 and $N-1$ having only one neighbour (communication is represented by arcs in the picture).

In the final position, each drone i must be at position $min + i\ell$.

This is a problem of formation control that can be dealt with along the lines exposed by Olfati-Saber *et al.* in [41], but with a significant difference: The protocols in [41] are assumed to be executed by all drones, but in the present case the two extreme drones are fixed at their initial positions, therefore the protocol examined in the present paper takes a different form.

Equation (8) in Section 3.1.2 applies to formation control problems, but in our case we want the second term of the right member to vanish, since in the final position each drone is equidistant from its two neighbours, so that (8) reduces to

$$\dot{x} = -\mathrm{L}\,x, \tag{9}$$

where each component x_i of vector x is the position of the i-th drone along the line segment.

However, equation (7), the discrete-time form of (5), does not apply; since the first and last drones are fixed, this is violating the basic assumption of the consensus protocol, i.e. that all drones have the same behaviour. Hence, the graph Laplacian and Perron matrices take the following forms:

$$L = \begin{pmatrix} \mathbf{0} & \mathbf{0} & 0 & 0 & \dots & 0 & 0 \\ -1 & 2 & -1 & 0 & \dots & 0 & 0 \\ 0 & -1 & 2 & -1 & \dots & 0 & 0 \\ 0 & 0 & -1 & 2 & \dots & 0 & 0 \\ & & & & \ddots & & \\ 0 & 0 & 0 & \dots & -1 & 2 & -1 \\ 0 & 0 & 0 & \dots & 0 & \mathbf{0} & \mathbf{0} \end{pmatrix}$$

$$\mathrm{P} = \begin{pmatrix} \mathbf{1} & \mathbf{0} & 0 & 0 & \dots & 0 & 0 \\ \epsilon & 1-2\epsilon & \epsilon & 0 & \dots & 0 & 0 \\ 0 & \epsilon & 1-2\epsilon & \epsilon & \dots & 0 & 0 \\ 0 & 0 & \epsilon & 1-2\epsilon & \dots & 0 & 0 \\ & & & & \ddots & & \\ 0 & 0 & 0 & \dots & \epsilon & 1-2\epsilon & \epsilon \\ 0 & 0 & 0 & \dots & 0 & \mathbf{0} & \mathbf{1} \end{pmatrix}$$

With the above matrices L and P, we have $\dot{x}_1 = \dot{x}_N = 0$ in the continuous-time model, and $x_1(k+1) = x_1(k)$, $x_N(k+1) = x_N(k)$ in the discrete-time model. The explicit form of the protocol is thus:

$$\begin{cases} x_1(k+1) &= x_1(k) \\ x_i(k+1) &= \epsilon x_{i-1}(k) + (1-2\epsilon)x_i(k) \\ & \quad +\epsilon x_{i+1}(k)\,, \\ & \qquad\qquad i \in [2\,..\,N-1] \\ x_N(k+1) &= x_N(k)\,. \end{cases} \tag{10}$$

Since the above protocol involves matrices L and P that have a different structure from the ones in [41], the results on protocol convergence from that paper do not apply. The rest of this section will show how the protocol can be specified with a higher-order theory, validated through co-simulation and verified by theorem proving.

The following subsections describe the different FMUs for the specific case study. The plant FMU is exported from OpenModelica, the controller FMU is a C program and the coordination FMU packages a PVSio interpreter executing the coordination algorithm.

5.1. The plant FMU

The linearized equations of a quadcopter have been implemented in an OpenModelica program, which simply expresses the equations for position and attitude in Section 3.1 in the Modelica language as shown in Fig. 7. More precisely, the translational dynamics are expressed as follows, where *psid* is the desired yaw angle ψ_d, *der* is the time derivative operator, g is the gravitational acceleration, *Kf* is a characteristic constant

```
parameter Real g = 9.81;
  parameter Real l = 0.3;
  parameter Real Kf = 1;
  parameter Real Km = 1;
  parameter Real m = 1;
  parameter Real Ixx = 0.001;
  parameter Real Iyy = 0.001;
  parameter Real Izz = 0.001;
  parameter Real x0=0;
  parameter Real y0=0;
  parameter Real z0=0;
  parameter Real psid_0=0;

  Real psid;
initial equation
  x = x0;
  y = y0;
  z = z0;
  phi = 0;
  theta = 0;
  psi = 0;
equation
  psid=psid_0;
  der(x) = dx;
  der(dx) = g * sin(psid) * phi + g * cos(psid) * theta;
  der(y) = dy;
  der(dy) = - (g * cos(psid) * phi) + g * sin(psid) * theta;
  der(z) = dz;
  der(dz) = 2 * sqrt(Kf * g / m) * (w1 + w2 + w3 + w4);
  der(phi) = dphi;
  der(dphi) = (l * sqrt(m * g * Kf) / Ixx) * (w2 - w4);
  der(theta) = dtheta;
  der(dtheta) = (l * sqrt(m * g * Kf) / Iyy) * (w3 - w1);
  der(psi) = dpsi;
  der(dpsi) = (Km * sqrt(m * g * Kf) / Izz) * (w1 - w2 + w3 - w4);
  annotation( ...);
end quadcopter;
```

FIGURE 7. Plant model in the OpenModelica editor.

of the rotor and m is the mass of the drone. Factor $2 \cdot sqrt(Kf \cdot g/m)$ is the C_z constant in equation (1).

```
psid = psid_0;
der(x) = dx;
der(dx) = g*sin(psid)*phi
          + g*cos(psid)*theta;
der(y) = dy;
der(dy) = -(g*cos(psid)*phi)
          + g*sin(psid)*theta;
der(z) = dz;
der(dz) = 2*sqrt(Kf*g/m)*(omega1 + omega2
             + omega3 + omega4);
```

The rotational dynamics are expressed as

```
der(phi) = dphi;
der(dphi) = (l*sqrt(Kf*m*g)/Ixx)
               *(omega2 - omega4);
der(theta) = dtheta;
der(dtheta) = (l*sqrt(Kf*m*g)/Iyy)
               *(omega3 - omega1);
der(psi) = dpsi;
der(dpsi) = (Km*sqrt(Kf*m*g)/Izz)
               *(omega1 - omega2
                 + omega3 - omega4);
```

where l is the distance of each rotor from the centre of mass, Ixx, Iyy and Izz are the axial moments of inertia and Km is another characteristic constant of the rotor. Factors $l \cdot sqrt(Kf \cdot m \cdot g)/Ixx$, $l \cdot sqrt(Kf \cdot m \cdot g)/Iyy$ and $Km \cdot sqrt(Kf \cdot m \cdot g)/Izz$ are the constants C_ϕ, C_θ and C_ψ, respectively, in the attitude equation.

The FMU is automatically generated from OpenModelica and allows the values of the parameters to be set before simulation.

5.2. The control FMU

For reasons of efficiency, the equations for the control (Section 3.1.1) have been implemented in a C function wrapped in a FMU. The FMU implements the FMI *doStep* function updating the values of $\omega_1, \ldots \omega_4$ according to (3) and (4).

For example, the code shown below computes the value of θ_c, according to equation (3), i.e. $\theta_c = -c_d(2\lambda_P \dot{x} + \lambda_P^2 e_x) - s_d(2\lambda_P \dot{y} + \lambda_P^2 e_y)$, with $c_d = (\cos \psi_d)/g$ and $s_d = (\sin \psi_d)/g$.

```
st->theta_c =
 - (cos(st->psi_d) / st->G)
   * (2*st->Lp * st->vel_x + st->Lp
     * st->Lp * (st->current_x - st->x_d))
 - (sin(st->psi_d)/st->G)
    * (2*st->Lp * st->vel_y + st->Lp
      * st->Lp  * (st->current_y - st->y_d));
```

In the code, *st* is a data structure storing variables and parameters, *Lp* stands for λ_P, *vel_x* and *vel_y* stand for $\dot{x}$ and $\dot{y}$, *(st->current_x - st->x_d)* and *(st->current_y - st->y_d)* stand for e_x and e_y, and *st->G* stands for g.

Since the controller on board of a drone is able to convert its dynamics into those of a single integrator, the solutions of equation (9) can be feasibly tracked by drones. The feedback control law compensates the non-linear terms of the dynamics and introduces those of the interaction with neighbours.

Moreover, when the numeric integration step τ is chosen small enough, and the controller output has a sufficient amplitude, the complete response of the drone, including the transient phase, is that of a first-order system.

5.3. The coordination FMU

```
coverage[N, n: posnat]: THEORY
BEGIN
drone_id: TYPE = below(N)
location: TYPE = [# x: real #]
state: TYPE+ =
    [#  self: location,
     id: drone_id #]

dmap: TYPE = [drone_id -> state]

systemState: TYPE =
    [# drones: dmap, timesteps: upto(n) #]
END coverage
```

LISTING 3. Type definitions for the coverage protocol.

The coverage protocol (10) has been expressed in two swarm-level PVS theories. Theory *coverage* (Listing 3) is based on *system* from Section 4.1. Type *location* is redefined as a single x coordinate, since the drones are assumed to be aligned. In the system state, the field for the communication topology has been dropped, since in this case each drone has only two communication links, assumed to be reliable and unchanging.

```
coverage_execution: THEORY
BEGIN IMPORTING initial

% Equation (7) for drone i
exec_cvg(a,b,c: real, i: drone_id): real =
  IF (i = 0) THEN min  ELSE
     IF (i = N - 1) THEN max
     ELSE eps*a + (1 - 2*eps)*b + eps*c
     ENDIF
  ENDIF

% selection of x coordinate of drone i
getx(s: systemState, i: drone_id): real =
    s`drones(i)`self`x

% update the position of every drone
exec_coverage(s: systemState): systemState =
    s WITH  [drones :=
        LAMBDA (i: drone_id):
        LET xp =
            IF i > 0 THEN getx(s, i - 1)
            ELSE min
            ENDIF,
            x = getx(s, i),
            xf =
            IF i < N - 1 THEN getx(s, i + 1)
            ELSE max
            ENDIF
        IN s`drones(i) WITH [self`x :=
exec_cvg(xp, x, xf, i)] ]

% invokes the protocol every simulation steps
tick(s: systemState) : systemState =
    IF (s`timesteps = n)
    THEN exec_coverage(s) WITH [timesteps := 0]
    ELSE s WITH [timesteps := s`timesteps + 1]
    ENDIF

% execution of the protocol
kth_step(k: nat): RECURSIVE systemState =
    IF (k = 0) THEN initial
    ELSE tick(kth_step(k - 1))
    ENDIF
    MEASURE k

END coverage_execution
```

LISTING 4. Coverage protocol in PVS.

Field *timestep* is a counter that is reset every ϵ seconds, and counts the current number of co-simulation steps after the last reset. Its value ranges on the interval [0..*n*] (*upto(n)*), where *n*, a parameter of the theory, equals ϵ/τ.

Theory *coverage_execution* (Listing 4) is patterned after *system_execution* from Section 4.1, and implements the function *kth_step* for the execution of the protocol. Function *exec_cvg* is the PVS form of (10), where *a*, *b* and *c* are the coordinates of the preceding, current and following drone, respectively. Function *getx* accesses the x-coordinate of the *i*-th drone in the system state. Function *exec_coverage* models the execution of the coverage protocol at every drone, updating the system map *drones* with the new positions of drones. The *drones* field is an anonymous function (a λ-expression) of a drone identifier. This function extracts the coordinates of drone *i* and of its neighbours from the swarm state (using function *getx*), and computes the new state of drone *i* according to the protocol.

Function *tick* is needed to synchronize the different Coordination FMUs, as explained in Section 4.3.

Function *kth_step* matches the function of the same name in Section 4.1, except that it does not call the protocol section directly, but through function *tick*. Definitions for global parameters and for the initial swarm state are contained in an additional theory, *initial* (not shown).

In the *initial* theory, constant *n* is defined as the ratio *eps*/*stepsize*, i.e. ϵ/τ. Every *n* timesteps, the new desired position of the drone is computed by a step of the algorithm (*exec_coverage*).

The above swarm-level theories provide a global view of the swarm state, assuming complete knowledge of each drone's state. A real drone, however, has no knowledge of the other drones, except for the information it receives from its neighbours. In order to perform a realistic co-simulation, a simpler theory, based on the state of a single drone, is needed.

Theory *fmu_coverage* (Listing 5) is a simplification of theory *coverage* above. In Theory *fmu_coverage*, the simulation state in the FMU is a projection of the swarm state to a single drone. The state of the drone is therefore implicit in the FMU, and it does not appear anymore as an argument of functions (see, for example, function *fmu_kth_step*). Fields *prec* and *foll* hold the location of the preceding and following drone, respectively, and their values are provided by the master algorithm of the co-simulation.

```
fmu_coverage[N: posnat]: THEORY
BEGIN
drone_id: TYPE = below(N)
location: TYPE = real

state: TYPE+ =
    [# prec: location, self: location,
    foll: location, id: drone_id #]

% projection of system state at a drone
simstate: TYPE =
    [# timesteps: integer, st: state #]
END fmu_coverage
```

LISTING 5. Type definitions for the executable coverage protocol.

Accordingly, Theory *fmu_exec_coverage* (Listing 6) is a simplification of *exec_coverage*. Equation (10) is expressed by function *fmu_exec_cvg* with the same algorithm as function *exec_cvg* above, whereas *fmu_tick* merges functions *exec_coverage* and *tick* above.

Function *fmu_kth_step* reproduces *kth_step*, replacing *tick* with *fmu_tick*.

```
fmu_exec_coverage: THEORY
BEGIN IMPORTING fmu_initial
fmu_exec_cvg(a,b,c: real, i: drone_id): real =
    IF (i = 0) THEN min
    ELSE
        IF (i = N - 1) THEN max
        ELSE eps*a + (1 - 2*eps)*b + eps*c
        ENDIF
    ENDIF

fmu_tick(s: simstate) : simstate =
    LET xp =
        IF (s`st`id > 0) THEN s`st`prec
        ELSE min
        ENDIF,
        x = s`st`self,
        xf =
        IF (s`st`id < N - 1) THEN s`st`foll
        ELSE max
        ENDIF
    IN
        COND
        s`timesteps >= eps/stepsize -> s
          WITH [st`self :=
              fmu_exec_cvg(xp, x, xf, s`st`id),
              timesteps := 0],
        s`timesteps < eps/stepsize -> s
          WITH [`timesteps := s`timesteps + 1]
        ENDCOND

fmu_kth_step(k: nat): RECURSIVE simstate =
    IF (k = 0) THEN initial
    ELSE fmu_tick(fmu_kth_step(k - 1))
    ENDIF
    MEASURE k
END fmu_exec_coverage
```

LISTING 6. Executable theory for the coverage protocol.

The FMUS involved in the co-simulation reported in Section 6 contain the simplified theories; the verifications performed in Section 7 rely on the swarm-level theories with a global view of the swarm state.

6. CO-SIMULATION RESULTS

In this section, the PVS theories introduced above are validated by simulation, i.e. they are used as an executable code that is packed in an FMU and integrated with other executable code (in C and Modelica) in a co-simulation environment. Simulation experiments corroborate confidence that the theories correctly model the behaviour of a drone swarm following the coverage protocol. The experiments consist in choosing a deployment scenario with a given initial placement of the swarm and observing if and how the swarm achieves the desired placement, with the drones uniformly spaced along the assigned line.

The coverage protocol has been simulated in a scenario of five quadcopters. Therefore, the co-simulation uses 15 FMUs, connected through the INTO-CPS COE as shown in Fig. 5. In the coordination algorithm, a fixed value has been chosen for the time discretization interval ϵ at which the desired position of each drone is updated, and a fixed value has been chosen for the co-simulation step size τ, corresponding to the frequency of communications between the low level control and the plant. Each drone has been tagged with an identifier number and the identifiers are ordered according to the initial position of the drones.

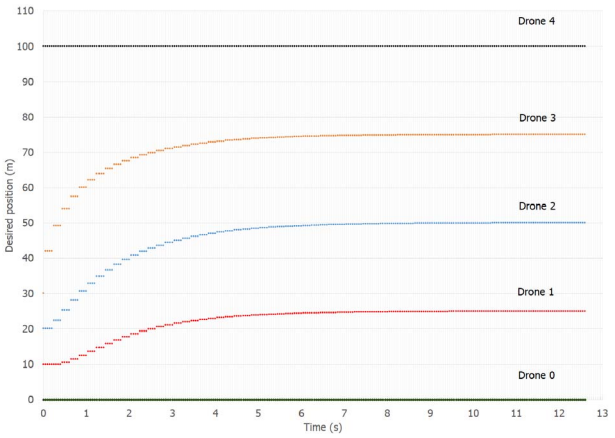

FIGURE 8. Plot of the desired position of drones with $\epsilon = 0.2$.

In the simulation two parameters are traced: the desired position and the actual position. The different behaviour between the desired positions, computed by the coordination algorithm, and the actual ones, reached by the drones, is due to the fact that the latter are the result of having included the drone dynamics and the control laws (3) in the co-simulation. This is the reason why a completely analytic approach is not viable. Co-simulation provides useful feedback for protocol analysis and validation.

The protocol in (10), expressed in PVS in Listings 5 and 6, has been simulated with the following initial positions: $x_1 = 0$, $x_2 = 10$, $x_3 = 20$, $x_4 = 30$ and $x_5 = 100$. The simulation parameters are $\epsilon = 0.2$ and $\tau = 0.05$.

As anticipated, parameter ϵ determines how often the desired position of each drone is updated. In the figures, ϵ is represented by the spacing between the vertical lines in the grid, i.e. each vertical line marks the completion of a simulation step. With $\epsilon = 0.1$, the desired position is constant for two co-simulation steps. With $\epsilon = 0.2$, the desired position is updated every four co-simulation steps (0.2 sec). With $\epsilon = 0.5$, the desired position is constant for ten co-simulation steps (0.5 s).

Figure 8 shows the evolution of the desired x for each drone. It can be seen that, at the first simulation step, the desired position of drone 4 increases, since that drone is farther from drone 5 than from drone 3, so it should get closer to drone 5, as per coordination protocol. In the initial configuration, drone 1 is instead equidistant from its neighbours, therefore its desired position in unchanged—similarly for drone 2.

After ϵ seconds, both drone 3 and drone 2 change their desired position: drone 2 is now farther from drone 3 than from drone 2.

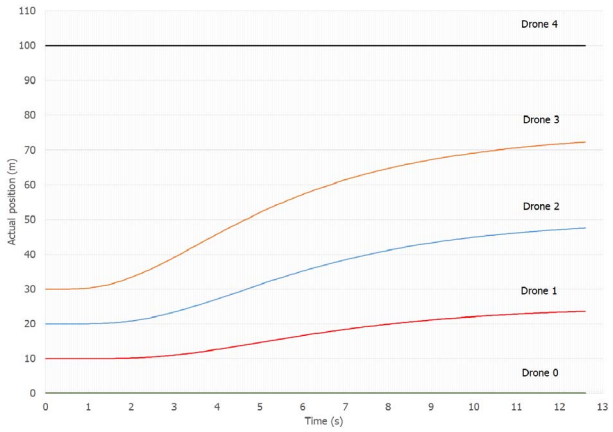

FIGURE 9. Plot of the actual position of the drones with $\epsilon = 0.2$.

Finally, after 3ϵ seconds (0.3 s), also drone 1 increases its desired position.

The desired positions asymptotically converge to the target positions $i\ell$ after four simulated seconds of co-simulation.

The actual positions of the drones for the same experiment is shown in Fig. 9. After five simulated seconds, the desired positions computed by the coordination algorithm (10) have reached the reference points (25, 50 and 75 for the moving drones with identifiers 1, 2 and 3, respectively) and at the end of the simulation the three moving drones reached their reference position. It can be seen that the drones follow the desired position and reach it after a delay.

Figure 10 shows the desired position of the central drone in the first 8 seconds of co-simulation, for different values of ϵ ($\epsilon \in \{0.1, 0.3, 0.5\}$). For higher values of ϵ, the co-ordination FMU updates the target positions less frequently and the difference between neighbours desired position increases. For this reason, the system does not converge for high values of ϵ.

A co-simulation was run using the same configuration of the previous examples, but choosing a higher value of ϵ than allowed by the verification results, namely $\epsilon = 0.7$. The result (Fig. 11) confirms that the algorithm does not converge.

The desired positions are recomputed every 12 co-simulation steps, they take very high values, and they may increase or decrease at successive steps. The values on the ordinates axis are one order of magnitude larger than in the previous cases. For example, the desired position of drone 3 after ϵ seconds is very close to 100 and it decreases at 2ϵ seconds. At that time, the desired position of drone 2 almost equals that of drone 3, and it decreases at the next synchronization point. As a result, the desired positions oscillate and after 3 seconds they may become negative.

Considering the physical system, in Fig. 12 we observe how the drones oscillate along the x axis. The oscillations become very large since the fifth second and, with the considered dynamics, drones 2 and 3 collide after 12 seconds. The oscillation of the actual position becomes noticeable later than the one of the desired positions. Again, this delay is determined by the drone dynamics.

Using the framework proposed in [44], a simple interface has been built, providing graphic feedback on the evolution of the co-simulation. The interface is an additional FMU that gathers the actual x and z positions of all the drones, showing them on the graphic interface. The desired z co-ordinate has a fixed value, and represents the final height for all the drones. This new FMU does not influence the behaviour of the drones. The graphic interface is shown in Fig. 13, with the drones in the initial positions. Figure 14 shows the position of drones at time 9.65 seconds. Video recordings of two simulation runs can be found in the github repository (https://github.com/mapalmieri/Drones). One video shows that the drone trajectories are unstable if the step size ϵ is outside the range specified in the hypotheses of Theorem 7.2 in Section 7.2 below.

As expected, the stability of the coordination protocol depends on the value of ϵ. As discussed in Section 7, a formal approach was used to find constraints on ϵ: First, a proof was attempted of a conjecture on system stability that took no assumptions on the value of ϵ; then, in the course of the proof it turned out that the conjecture could be proved only by adding constraints on ϵ.

Finally, we note that one of the advantages of co-simulation is the reusability of models. The *control FMU* and the *plant FMU* can be used in conjunction with different co-ordination protocols. For example, the co-simulation multi-model was validated with an implementation of the original consensus protocol from [41], where all the drones converge toward a single target, i.e. the average of their initial positions (Section 3.1.2). The PVS theory for the consensus protocol can be easily derived by rewriting the *exec_cvg* function in theory *exec_coverage*, according to Equation (5). This PVS specification was used to build the new FMU that was plugged back into the multi-model.

A co-simulation was then run with initial positions $x_0 = 0$, $x_1 = 10$, $x_2 = 20$, $x_3 = 30$ and $x_4 = 100$, and simulation parameters $\epsilon = 0.3$ and $\tau = 0.05$. After 10 simulated seconds, all the drones had the desired x coordinate of 32, equal to the average of the initial positions, in accordance with the theory in [41]. The co-simulation results are shown in Figs 15 and 16.

7. FORMAL VERIFICATION

While the previous section deals with the validation of the PVS theories for the coverage protocol by simulation of a particular scenario, this section shows how properties of the protocol can be proved in a formal and general way. More precisely, this section presents results on the formal verification of the coverage protocol (10) in Section 5, using the abstract theory *exec_coverage* introduced in Section 5.3.

The main result is that the positions of the drones asymptotically approach the desired configuration (Theorem 7.8),

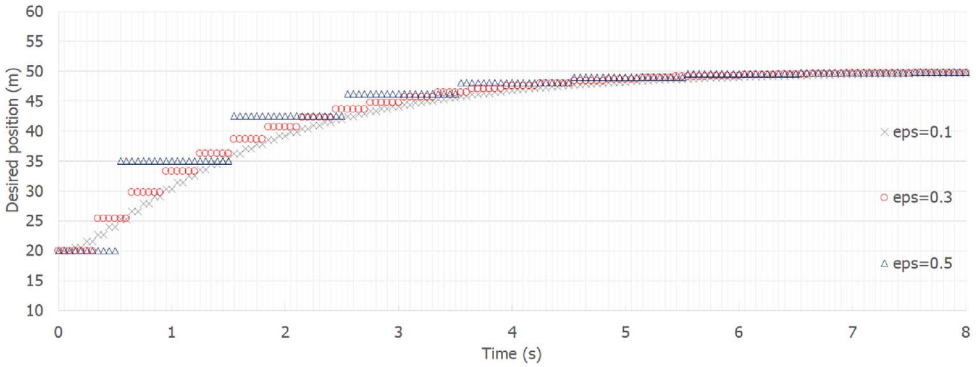

FIGURE 10. Desired position of the central drone, for different values of ϵ: 0.1, 0.3 and 0.5seconds.

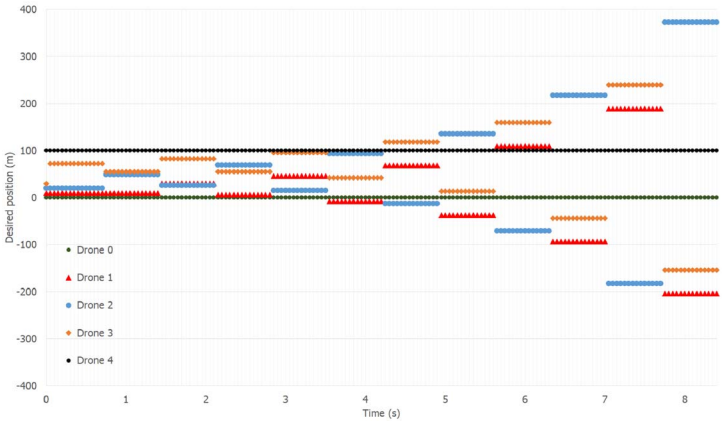

FIGURE 11. Plot of the desired x of the drones with $\epsilon = 0.7$, coverage protocol.

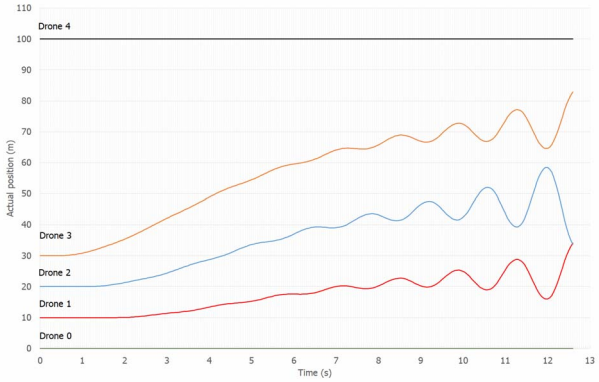

FIGURE 12. Plot of the actual x of the drones with $\epsilon = 0.7$, coverage protocol.

under the assumptions stated in the following and an additional condition on epsilon. Fundamental in the proof of *convergence* are the following properties: the trajectories of drones do not cross (Theorem 7.2); the target position of each drone is an upper bound for the position at each step (Theorem 7.3); the sums of the coordinates of drones at each step are a non decreasing sequence (Theorem 7.5); the sum of the coordinates of drones may remain constant for some number of steps, but will eventually increase unless all drones have reached their target position (Theorem 7.6, Theorem 7.7).

For better readability, a standard mathematical notation is used in Section 7.1 and part of Section 7.2. The PVS syntax is used in Section 7.2 to show parts of a proof, and in the Appendix to show a more exhaustive example of actual interactive theorem proving.

7.1. Definitions and assumptions

For simplicity, it is assumed that the segment to be spanned by the drones starts at the origin of the x axis, i.e. $min = 0$ and $max = d$, where d the length of the segment. In the following, h, i, j and k denote natural numbers. Moreover, from now on, X refers to the position of drones, and its usage is defined more clearly in the definition below.

The following definitions will be used:

1. $N \geq 3$ is the total number of drones, numbered starting from 0;

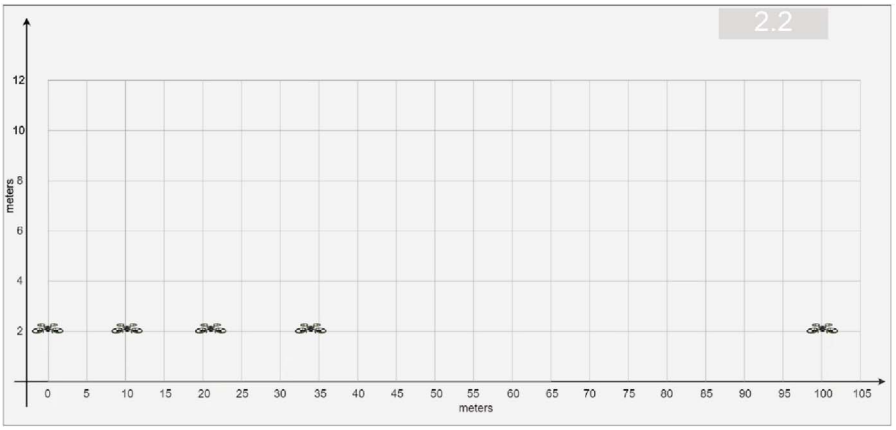

FIGURE 13. Screenshot of the simulation interface (t = 2.2 seconds).

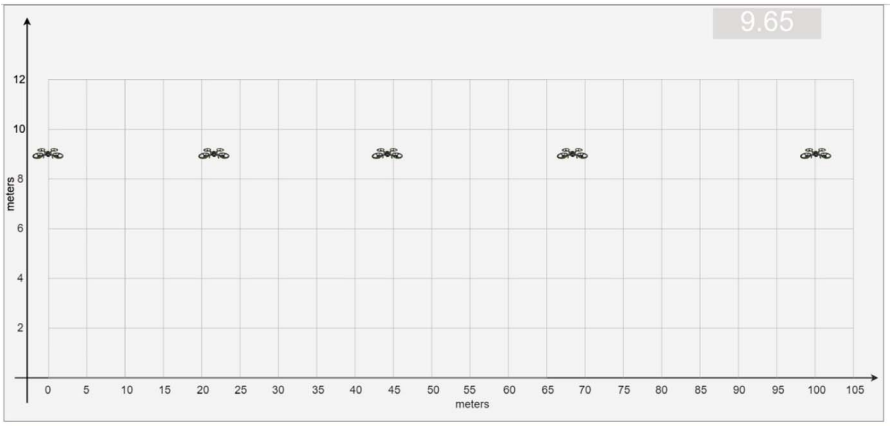

FIGURE 14. Screenshot of the simulation interface (t = 9.65 seconds).

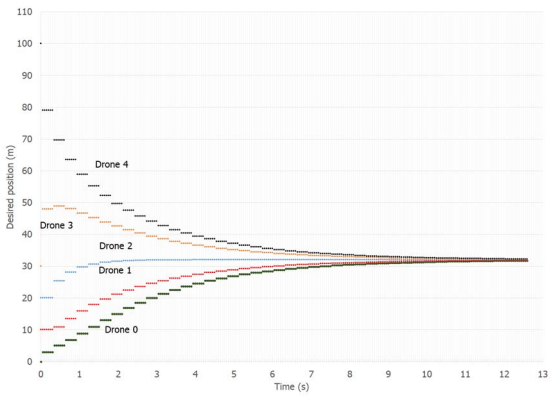

FIGURE 15. Plot of the desired position of drones with $\epsilon = 0.3$ for the original consensus protocol.

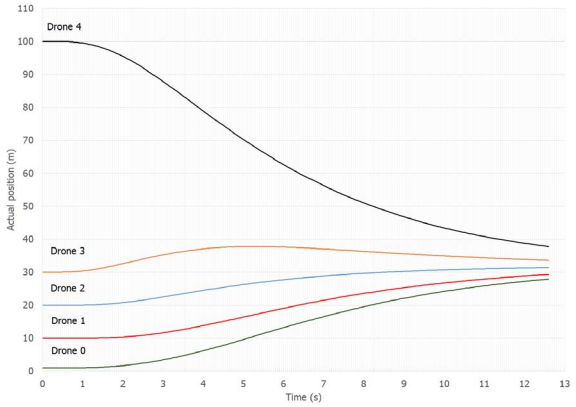

FIGURE 16. Plot of the actual position of the drones with $\epsilon = 0.3$ for the original consensus protocol.

2. $\ell = d/(N-1)$ is the desired distance between adjacent drones;
3. $\bar{X}_i = i\ell$ is the target position of drone i;
4. $X^k = (X_0^k, \cdots, X_{N-1}^k)$ is the placement of the drones at step k, with X_i^k the position of drone i at step k;
5. the initial placement is X^0;

6. $\epsilon \in [0, 1]$ is the step size;
7. $c^k = \sum_{i=0}^{N-1} X_i^k$ is the sum of the drone coordinates at step k.

With the above definitions, we make the following assumptions:

1. the first and last drones maintain a fixed position at $X_0 = 0$ and $X_{N-1} = d$, respectively;
2. the drones are numbered in order of increasing distance from X_0, i.e. $\forall_{i<N-1} X_i^0 < X_{i+1}^0$, and the drones with index $i \in [1 \,..\, N-2]$ will be called *internal*;
3. initially, each internal drone is placed within the first subsegment: $\forall_{i\in[1\,..\,N-2]} X_i^0 < \ell$. This is a simplifying assumption for Theorem 7.6.
4. at each step $k+1$, the position of each internal drone i is related to the positions of drone i and its neighbours at the previous step by the recurrence relation: $X_i^{k+1} = \epsilon X_{i-1}^k + (1-2\epsilon) X_i^k + \epsilon X_{i+1}^k$

For this specific problem, it is necessary to assume that all the drones are initially placed in the first subsegment and arranged in order by their indices. In the course of the proof, the user finds that further assumptions are needed. In the specific case, a constraint on ϵ is suggested by the unprovability of certain subgoals.

The position of the i-th drone at the k-th step is then X_i^k, which is expressed in the PVS theory as the composition of *kth_step* and *exec_coverage*, respectively, abstracting from details related to synchronization and record-based data representation.

7.2. Proofs and results

This section collects results from formal verification of the theories developed to model the coverage protocol, and shows how the process of interactive proof can lead to the formulation of constraints on design parameters. An example of how an interactive proof is carried out in practice is shown in the Appendix.

With the previously introduced assumptions points from 1 to 4 above, we want to prove that the drones maintain their relative spatial ordering, i.e. their trajectories do not cross. We start by attempting to prove the following conjecture, where no assumption on parameter ϵ is made:

CONJECTURE 7.1. (INVARIANT SPATIAL ORDERING) $\forall_{k\geq 0} \forall_{i\in[0\,..\,N-2]} X_i^k < X_{i+1}^k$.

Excerpts from the proof trace for Conjecture 7.1 are shown below. The trace is a linearized display of a proof tree where each node is a sequent, with a horizontal dashed line separating antecedents (above) and consequents (below). Each formula in the sequent is labelled with a number, negative for antecedents and positive for consequents. Each sequent (except for the first one and those at the beginning of a branch) is preceded by the PVS rule applied by the user to the previous sequent, a synthetic description of the rule, and by a sequent label. The label is the name of the formula (*goal*) to be proved, and the same name adorned with numbers identifies derived subgoals. In the following trace, *no_cross* is the label of the initial goal, *no_cross.1* and *no_cross.2* are the labels of the first two subgoals.

The PVS code for the axiom on the drones' initial position and for Conjecture 7.1 is shown in Listing 7.

```
init_ax: AXIOM
  FORALL (i: subrange(N-1)):
    initial(i) < initial(i+1);

no_cross: THEOREM
  FORALL (k: nat):
    FORALL (i:{n:nat | n < N-1}):
        kth_step(k)(i) < kth_step(k)(i+1)
```

LISTING 7. Theorem 7.2 (no_cross) in PVS.

The first step is the *induct* command that causes the theorem prover to start a proof by induction on k, generating the base case *no_cross.1* and the induction step *no_cross.2*:

```
Rule? (induct k)
no_cross.1 :

  |-------
{1} FORALL (i: {n: nat | n < N - 1}):
       kth_step(0) (i)
        < kth_step(0) (i + 1)
```

The base case is proved in few steps using the axiom *init_ax* corresponding to Definition 5 and Assumption 2, added to the antecedents with the *lemma* rule.

```
Rule? (skolem 1)
no_cross.1 :

  |-------
{1}   kth_step(0)(i) < kth_step(0)(i + 1)
```

Expanding *kth_step* and importing the axiom *init_ax* lead to:

```
Rule? (rewrite "kth_step")
no_cross.1 :

{-1}  FORALL (i: below(N - 1)):
       initial(i) < initial(i + 1)
  |-------
[1]   initial(i) < initial(i + 1)
```

which is automatically proved after instantiating the top quantifier in {-1} with the term i. This completes the proof of `no_cross.1`.

The induction step is as follows:

```
no_cross.2 :

  |-------
{1} FORALL j:
       (FORALL (i: {n: nat | n < N - 1}):
         kth_step(j) (i)
          < kth_step(j) (i + 1)
        IMPLIES
         (FORALL (i: {n: nat | n < N - 1}):
           kth_step(j + 1) (i)
            < kth_step(j + 1) (i + 1)
```

Applying a simple rearrangement of the goal (with *skolem* and *flatten*):

```
Rule? (flatten)
no_cross.2 :
{-1}  FORALL (i: {n: nat | n < N - 1}):
         kth_step(j)(i) < kth_step(j)(i + 1)
  |-------
{1}   FORALL (i: {n: nat | n < N - 1}):
       kth_step(j + 1)(i) <
            kth_step(j + 1)(i + 1)
```

After some manipulations and expanding `kth_step`, the following sequent is obtained:

```
Rule? (rewrite "kth_step" 1)
no_cross.2 :

[-1]  N >= 3
[-2]  FORALL (i: {n: nat | n < N - 1}):
       kth_step(j)(i) < kth_step(j)(i + 1)
  |-------
{1}   IF (i = 0)
      THEN
        IF (1 + i = N - 1) THEN min < max
        ELSE min < kth_step(j)(1 + i)
                   - 2*(kth_step(j)(1 + i)*eps)
                   + eps*kth_step(j)(2 + i)
                   + eps * kth_step(j)(i)
        ENDIF
      ELSE kth_step(j)(i)
           - 2*(kth_step(j)(i)*eps)
           + eps*kth_step(j)(i - 1)
           + eps*kth_step(j)(1 + i)
           <
           IF (1 + i = N - 1) THEN max
           ELSE kth_step(j)(1 + i)
                - 2*(kth_step(j)(1 + i)*eps)
                + eps*kth_step(j)(2 + i)
                + eps * kth_step(j)(i)
           ENDIF
      ENDIF
```

Repeated applications of the *split* command, together with sequent rearrangement and simplifications, yield four subgoals corresponding to the following cases:

- $i = 0$ and $i + 1 = N - 1$: this implies that there are only two drones, which contradicts the hypothesis $N \geq 3$;
- $i = 1$ and $i + 1 \neq N - 1$, the second drone;
- $i \neq 0$ and $i + 1 = N - 1$, the second-but-last drone.
- $i \neq 0$, $i \neq 1$, and $(i + 1 < N - 1)$, the other internal drones.

The first three are special cases that are proved by manipulating expressions. In the fourth case, mathematical manipulations lead to:

```
[-1]  eps > 0
[-2]  N >= 3
[-3]  FORALL (i: {n: nat | n < N - 1}):
       kth_step(j)(i) < kth_step(j)(1 + i)
  |-------
{1}   kth_step(j)(i) * (1 - 3 * eps) <=
         kth_step(j)(1 + i) * (1 - 3 * eps)
```

Dividing by $(1 - 3 * eps)$ leads to:

```
Rule? (cancel-by "1 - 3 * eps" -1)
no_cross.2.2.2.1.1.1.1.5.2 :

{-1}  eps > 0
{-2}  N >= 3
{-3}  FORALL (i: {n: nat | n < N - 1}):
       kth_step(j)(i) < kth_step(j)(1 + i)
  |-------
{1}   (1 - 3 * eps) > 0
{2}   kth_step(j)(1 + i) <= kth_step(j)(i)
```

Consequent {2} is false because it contradicts the antecedent $\{-3\}$. Therefore, in order to continue the proof it is necessary to prove that the consequent {1} is true. This requires the assumption that $\epsilon < \frac{1}{3}$.

With the newly found constraint on ϵ, the following properties are proved.

THEOREM 7.2. (INVARIANT SPATIAL ORDERING). *Drones maintain their relative spatial ordering, i.e. their trajectories do not cross:*

If $\epsilon < 1/3$, *then* $\forall_{k\geq 0}\forall_{i\in[0\,..\,N-2]} X_i^k < X_{i+1}^k$.

THEOREM 7.3. (REFERENCE POSITION UPPER BOUND) *At each step, the target position of each drone is an upper bound for its current position:*

If $\epsilon < 1/2$, *then* $\forall_{k\geq 0}\forall_{i\in[1\,..\,N-2]} X_i^k \leq i\ell$.

A corollary of the above theorem states properties of c^k (the sum of drone co-ordinates at step k). The properties of the sum of drone positions at each step are used in further proofs.

COROLLARY 7.4. (i) c^k is bounded: $\forall_{k\geq 0} \quad c^k \leq \frac{N(N-1)}{2}\ell$
(ii) c^k equals $\frac{N(N-1)}{2}\ell$ if and only if all drones are at their target position: $\forall_{k\geq 0}(c^k = \frac{N(N-1)}{2}\ell \iff \forall_i X_i^k = i\ell)$

THEOREM 7.5 (NON-DECREASING). *The sums of the drone coordinates at each step (i) are a nondecreasing sequence, and (ii) if all drones are at their target position, then $c^{k+1} = c^k$:*
(i) If $\epsilon < 1/2$, then $\forall_{k\geq 0} c^{k+1} \geq c^k$;
(ii) $\forall_i X_i^k = i\ell \implies c^{k+1} = c^k$.

The following theorem establishes a relationship between the sequence in time (i.e. wrt simulation steps) of the sums of drone coordinates and the drone positions at each step. This relationship is used in further theorems.

THEOREM 7.6 (CONVERGENCE). *Under assumption 3 and the constraint $\epsilon < \frac{1}{3}$, we have that (i) for any step k, if all drones are at their target position, then the sum of their co-ordinates does not change at any further step, and (ii) if the sum of drone coordinates remains constant after a given step k, then all drones are at their target positions at step k:*
(i) $\forall_k((\forall_i X_i^k = i\ell) \implies \forall_{h\geq 1} c^{k+h} = c^k)$,
(ii) $\forall_k(\forall_{h\geq 1} c^{k+h} = c^k) \implies \forall_i X_i^k = i\ell)$.

The following Theorem 7.7 states that the sum of the drone coordinates may remain constant for some number of steps, but will eventually increase unless all drones have reached their target position.

THEOREM 7.7 (GLOBALLY INCREASING). *If there is a step k such that $c^{k+1} = c^k$ and one or more drones are not at the target position, then there is a number $h > 1$ such that $c^{k+h+1} > c^{k+h}$:*
$(\exists_k(c^{k+1} = c^k \wedge \exists_{j\geq 3}(X_j^k < i\ell \wedge \forall_{i<j} X_i^k = i\ell))) \implies \exists_{h\geq 1} c^{k+h+1} > c^{k+h}$

We can finally prove that the position of each drone approaches monotonically the drone's target position as k increases.

THEOREM 7.8 (LIMIT). $\forall_{i\in[0\,..\,N-1],k\in\mathbb{N}} \lim_{k\to\infty} X_i^k = i\ell$.

The PVS proof of the invariant spatial ordering theorem (Theorem 7.2) is shown in the Appendix.

8. DISCUSSION AND CONCLUSIONS

The approach to co-simulation and verification proposed in this paper consists essentially in modelling the control parts of a CPS with one or more PVS theories and the physical parts (the plant) with a domain-specific language. The same PVS model is used both for simulation (i.e. validation) and formal proof (i.e. verification).

In the case at hand, this general principle has been applied as follows: (i) the distributed consensus protocol is modelled as a PVS theory; (ii) the control part in charge of steering a single drone is implemented in C; and (iii) the plant of a single drone is modelled in Modelica.

The choice of using two different languages (a higher-order logic language and an imperative one) for the control part is motivated by the fact that the goal of this case study is validating and verifying the consensus protocol, whereas the control of a single drone is a well-known problem and the correctness of the adopted algorithm has been proved. Alternatively, the control part of a single drone could be done in PVS and lumped with the distributed consensus protocol, or it could be integrated in the Modelica model. The adopted architecture, using three distinct models, is arguably more modular.

In this way, the OpenModelica simulator and the C implementation of the drone control afford an efficient execution of the computation-intensive simulation of drone dynamics, while the PVSio interpreter uses the same model of the consensus protocol as the one used for verification, thus enabling a cross-check between simulation and verification.

A further advantage of the co-simulation approach is that it makes it possible to simulate models produced by different tools (e.g. OpenModelica and 20-sim) possibly acquired off-the-shelf from different providers, requiring only the generation of the appropriate FMUs.

This approach is quite general and its application to the case study shows some advantages of combining formal verification and co-simulation in the analysis of CPSs. FMI-based co-simulation facilitates the decomposition of a large, complex model into submodels along natural lines, i.e. according to their function or physical model, and developers are free to choose how to apply formal verification to each submodel. For example, higher-order logic could be used for the control part and dynamic logic for the physical plant. It is also possible to focus formal verification on a single, more critical submodel, as in this case study. Using a co-simulation standard such as FMI also simplifies communication among the different teams involved in the design phase.

The approach discussed in this paper is applicable to a large class of systems, including cyber-physical ones, thanks to the expressiveness of higher-order logic, the deductive power of sequent calculus and the cumulative expertise of the theorem-proving community, which is concretely available in the form of published literature and verified theories. The latter, in particular, span many areas of mathematics, from basic ones such as real analysis, trigonometry and linear algebra, to more specialized topics such as various classes of polynomials or interval arithmetic.

It would be naïve to claim that higher-order logic theorem proving could dramatically ease the effort of control system design for CPS. Such an endeavour poses hard and evolving challenges requiring hard work and ingenuity, but it is safe to expect theorem proving to make a solid foundation to deal effectively with them (no silver bullet, but well-honed tools). For example, available NASALIB theories on multivariate polynomials can be used in optimization problems

arising in model-predictive control design. Theories on interval and affine arithmetic can be used to verify that a given numerical method guarantees the precision needed in systems subject to bifurcation and other numerical instability problems.

As examples of application to simple non-linear control prob lems, PVS has been used to verify safety properties in a non-linear hybrid system [6] using over- and under-approximations, and to compute an upper bound to the maximum distance from a target line for a line seeking robot [4].

Further, generally applicable theories or theory patterns are being developed in the area of linearized control systems [16].

Finally, it would be convenient to apply this approach to the co-operative systems represented by the linear time-varying models typical of opinion dynamics systems [21]. Indeed, since in these systems the neighbourhood changes discontinuously with time, an analytic direct analysis is known to be a hard task.

8. APPENDIX

In the following, the PVS proof of invariant spatial ordering theorem (Theorem 7.2) is shown.

```
init_ax: AXIOM
FORALL (i:subrange(N-1)):
    initial(i) < initial(i+1) ;
spatial_ordering_invariant: THEOREM
eps <= 1/3  IMPLIES
FORALL (k: nat):
    FORALL (i:{ n:nat | n < N-1}):
        kth_step(k)(i) < kth_step(k)(i+1)
```

LISTING 8. Theorem 7.1 (spatial order invariant) in PVS.

The proof started by the theorem shown in Listing 8. The first step is a simple rearrangement (*flattening*) of the goal: no_cross:

```
no_cross :

  |-------
  {-1}  eps <= 1 / 3 IMPLIES
     (FORALL (k: nat):
       FORALL (i: {n: nat | n < N - 1}):
        getx(kth_step(k), i)
         < getx(kth_step(k), i + 1))

Rule? (flatten)
Applying disjunctive simplification to
flatten sequent, this simplifies to:
no_cross :

{-1}  eps <= 1 / 3
  |-------
{1} FORALL (k: nat):
       FORALL (i: {n: nat | n < N - 1}):
        getx(kth_step(k), i)
         < getx(kth_step(k), i + 1)
```

Then, the *induct* command causes the theorem prover to start a proof by induction on *k*, generating the base case *no_cross.1* and the induction step *no_cross.2*:

```
Rule? (induct k)
Inducting on k on formula 1,
this yields  2 subgoals:
no_cross.1 :

[-1]  eps <= 1 / 3
  |-------
{1} FORALL (i: {n: nat | n < N - 1}):
       getx(kth_step(0), i)
        < getx(kth_step(0), i + 1)
```

The base case is proved in few steps (not shown) using two axioms (not shown) corresponding to Definition 5 and Assumption 2, added to the antecedents with the *lemma* rule.

The induction step is:

```
no_cross.2 :

[-1]  eps <= 1 / 3
  |-------
{1} FORALL j:
       (FORALL (i: {n: nat | n < N - 1}):
         getx(kth_step(j), i)
          < getx(kth_step(j), i + 1))
        IMPLIES
         (FORALL (i: {n: nat | n < N - 1}):
           getx(kth_step(j + 1), i)
            < getx(kth_step(j + 1), i + 1))
```

The outermost quantifier is eliminated by the *skolem* command, which replaces the occurrences of the induction variable *j* in formula {1} with the new constant *j*, which has been chosen to have the same name of the variable it replaces (another name could have been chosen, or generated by the theorem prover):

```
Rule? (skolem 1 j)
For the top quantifier in 1, we introduce
Skolem constants: j, this simplifies to:
no_cross.2 :

[-1]  eps <= 1 / 3
  |-------
{1} (FORALL (i: {n: nat | n < N - 1}):
       getx(kth_step(j), i)
        < getx(kth_step(j), i + 1))
      IMPLIES
       (FORALL (i: {n: nat | n < N - 1}):
         getx(kth_step(j + 1), i)
          < getx(kth_step(j + 1), i + 1))
```

After flattening and skolemizing again, we obtain:

```
no_cross.2 :

[-1] FORALL (i: {n: nat | n < N - 1}):
       getx(kth_step(j), i)
        < getx(kth_step(j), i + 1)
[-2] eps <= 1 / 3
  |-------
{1} getx(kth_step(j + 1), i)
       < getx(kth_step(j + 1), i + 1)
```

where *j* is a constant in formulas [-1] and {1}, while the symbol *i* represents a variable in [-1] and a constant in {1}. The definition of *kth_step* is expanded in {1}, and then the one of *tick*:

```
Rule? (rewrite kth_step 1)
...
Rule? (expand tick 1 1)
Expanding the definition of tick,
this simplifies to:
no_cross.2 :

[-1] FORALL (i: {n: nat | n < N - 1}):
       getx(kth_step(j), i)
        < getx(kth_step(j), i + 1)
[-2] eps <= 1 / 3
  |-------
{1} getx(IF (kth_step(j)`timesteps = n)
       THEN exec_coverage(kth_step(j)) WITH
        [timesteps := 0]
       ELSE kth_step(j) WITH
        [timesteps :=
         1 + kth_step(j)`timesteps]
       ENDIF,
       i)
       < getx(tick(kth_step(j)), 1 + i)
```

The first argument of *getx* is a conditional expression. The *lift-if* command lifts the conditional to the outermost level:

```
Rule? (lift-if)
Lifting IF-conditions to the top level,
this simplifies to:
no_cross.2 :

[-1] FORALL (i: {n: nat | n < N - 1}):
       getx(kth_step(j), i)
        < getx(kth_step(j), i + 1)
[-2] eps <= 1 / 3
  |-------
{1} IF (kth_step(j)`timesteps = n)
      THEN getx(exec_coverage(kth_step(j))
       WITH [timesteps := 0], i)
       < getx(tick(kth_step(j)), 1 + i)
      ELSE getx(kth_step(j)
       WITH [timesteps :=
              1 + kth_step(j)`timesteps], i)
       < getx(tick(kth_step(j)), 1 + i)
      ENDIF
```

The *split* command expands the conditional in two complementary implications, i.e.,

```
(kth_step(j)`timesteps = n) IMPLIES
 getx(exec_coverage(kth_step(j))
  WITH [timesteps := 0], i)
  < getx(tick(kth_step(j)), 1 + i)
```

and

```
NOT (kth_step(j)`timesteps = n) IMPLIES
 getx(kth_step(j)
  WITH [timesteps :=
         1 + kth_step(j)`timesteps], i)
  < getx(tick(kth_step(j)), 1 + i)
```

thus yielding two subgoals. The first one has the more complex subproof, since it represents the case when the system state is updated:

```
no_cross.2.1 :

[-1] FORALL (i: {n: nat | n < N - 1}):
       getx(kth_step(j), i)
        < getx(kth_step(j), i + 1)
[-2] eps <= 1 / 3
  |-------
{1} (kth_step(j)`timesteps = n) IMPLIES
       getx(exec_coverage(kth_step(j))
        WITH [timesteps := 0], i)
        < getx(tick(kth_step(j)), 1 + i)
```

The complexity of the proof is mainly due to the structure of the functions involved, but the proof strategy is quite simple. First, the sequent is reduced to the form

```
no_cross.2.1

[-1] (kth_step(j)`timesteps = n)
[-2] FORALL (i: {n: nat | n < N - 1}):
       getx(kth_step(j), i)
        < getx(kth_step(j), i + 1)
[-3] eps <= 1 / 3
  |-------
{1} getx(exec_coverage(kth_step(j)), i)
       < getx(tick(kth_step(j)), 1 + i)
```

Then, the inequality in {1} is transformed by repeated expansions of *x_coverage* and *tick*. This produces many branches due to the conditional expressions in the functions. In each branch, the resulting inequalities are solved by instantiating the induction hypothesis [1] and doing algebraic manipulations.

For example, the following sequent:

```
no_cross.2.1.1.1.1.1.1 :

{-1} 1 + i < N - 1
[-2] i > 0
[-3] (kth_step(j)`timesteps = n)
[-4] FORALL (i: {n: nat | n < N - 1}):
       getx(kth_step(j), i)
        < getx(kth_step(j), i + 1)
[-5] eps <= 1 / 3
  |-------
{1} exec_cvg(getx(kth_step(j), i - 1),
        getx(kth_step(j), i),
        getx(kth_step(j), 1 + i), i)
        < exec_cvg(getx(kth_step(j), i),
            getx(kth_step(j), 1 + i),
            getx(kth_step(j), 2 + i), 1 + i)
```

is on the branch corresponding to the conditions *kth_step(j)* *`timesteps* $= n$, $i > 0$, and $i < N - 2$. Expanding *exec_cvg* yields, with a couple of transformations,

```
no_cross.2.1.1.1.1.1.1 :

[-1] 1 + i < N - 1
[-2] i > 0
[-3] (kth_step(j)`timesteps = n)
{-4} FORALL (i: {n: nat | n < N - 1}):
        getx(kth_step(j), i)
         < getx(kth_step(j), 1 + i)
{-5} eps <= 1/3
  |-------
{1} getx(kth_step(j), i)
      - 2*(getx(kth_step(j), i)*eps)
      + eps*getx(kth_step(j), i - 1)
      + eps*getx(kth_step(j), 1 + i)
       < getx(kth_step(j), 1 + i)
         - 2*(getx(kth_step(j), 1 + i)*eps)
         + eps*getx(kth_step(j), 2 + i)
         + eps*getx(kth_step(j), i)
```

The induction hypothesis {4} is then instantiated successively with $i - 1$, i, and $i + 1$, obtaining a sequent with the three inequalities [-5], [-6], and [-7] in the antecedent:

```
no_cross.2.1.1.1.1.1.1 :

[-1] 1 + i < N - 1
[-2] i > 0
[-3] (kth_step(j)`timesteps = n)
[-4] FORALL (i: {n: nat | n < N - 1}):
       getx(kth_step(j), i)
        < getx(kth_step(j), 1 + i)
{-5} getx(kth_step(j), i + 1)
       < getx(kth_step(j), 1 + (i + 1))
[-6] getx(kth_step(j), i)
       < getx(kth_step(j), 1 + i)
[-7] getx(kth_step(j), i - 1)
       < getx(kth_step(j), 1 + (i - 1))
[-8] eps <= 1/3
  |-------
[1] getx(kth_step(j), i)
    - 2*(getx(kth_step(j), i)*eps)
    + eps*getx(kth_step(j), i - 1)
    + eps*getx(kth_step(j), 1 + i)
     < getx(kth_step(j), 1 + i)
       - 2*(getx(kth_step(j), 1 + i)*eps)
       + eps*getx(kth_step(j), 2 + i)
       + eps*getx(kth_step(j), i)
```

Further manipulations and some utility lemmas (not shown) make it possible to assemble an antecedent formula matching the consequent, thus solving the subgoal:

```
no_cross.2.1.1.1.1.1.1.1.1.1 :

[-1]  getx(kth_step(j), i)
      - 3*eps*getx(kth_step(j), i)
       < getx(kth_step(j), 1 + i)
         - 3*eps*getx(kth_step(j), 1 + i)
[-2]  getx(kth_step(j), i - 1)*eps
       < getx(kth_step(j), 2 + i)*eps
[-3]  1 + i < N - 1
[-4]  i > 0
[-5]  (kth_step(j)`timesteps = n)
[-6]  FORALL (i: {n: nat | n < N - 1}):
        getx(kth_step(j), i)
         < getx(kth_step(j), 1 + i)
[-7]  getx(kth_step(j), 1 + i)
         < getx(kth_step(j), 2 + i)
[-8]  getx(kth_step(j), i)
         < getx(kth_step(j), 1 + i)
[-9]  getx(kth_step(j), i - 1)
         < getx(kth_step(j), i)
[-10] eps <= 1/3
  |-------
[1] getx(kth_step(j), i - 1)*eps
    + getx(kth_step(j), i)
    - 3*getx(kth_step(j), i)*eps
     < getx(kth_step(j), 2 + i)*eps
       + (getx(kth_step(j), 1 + i))
       - 3*getx(kth_step(j), 1 + i)*eps

Rule? (add-formulas -1 -2)
Adding formulas -1 and -2,

This completes the proof of
no_cross.2.1.1.1.1.1.1.1.1.1.
```

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their useful comments and suggestions. The authors would also like to thank the INTO-CPS association for providing the co-simulation environment.

FUNDING

Ministero dell'Istruzione, dell'Università e della Ricerca, Italy, in the framework of the CrossLab project (Departments of Excellence).

DATA AVAILABILITY

The data underlying this article will be shared on reasonable request to the corresponding author.

REFERENCES

[1] Alur, R. and Dill, D.L. (1994) A theory of timed automata. *Theor. Comput. Sci.*, 126, 183–235.

[2] Attarzadeh Niaki, S.H. and Sander, I. (2011) Co-simulation of Embedded Systems in a Heterogeneous MoC-Based Modeling Framework. In *The 6th IEEE Int. Symposium on Industrial and Embedded Systems*, NW Washington, DC, USA, June 15–17, 2011, pp. 238–247. IEEE Computer Society, Västerås, Sweden.

[3] Behrmann, G., David, A., Larsen, K.G., Hakansson, J., Petterson, P., Wang, Y. and Hendriks, M. (2006) UPPAAL 4.0. In *The 3rd Int. Conf. on Quantitative Evaluation of Systems (QEST 2006)*, NW Washington, DC, USA, Sept. 11–14, 2006, pp. 125–126. IEEE Computer Society, Riverside, CA, USA.

[4] Bernardeschi, C., Domenici, A., Fagiolini, A. and Palmieri, M. (2020) Block-Based Models and Theorem Proving in Model-Based Development. In *The 2nd Interactive Workshop on the Industrial Application of Verification and Testing, ETAPS 2020 Workshop (InterAVT 2020)*, Virtual event, 25 April, 2020, pp. 1–8.

[5] Bernardeschi, C., Dini, P., Domenici, A., Palmieri, M. and Saponara, S. (2020) Formal verification and co-simulation in the design of a synchronous motor control algorithm. *Energies*, 13, 1–23.

[6] Bernardeschi, C. and Domenici, A. (2016) Verifying safety properties of a nonlinear control by interactive theorem proving with the prototype verification system. *Inf. Process. Lett.*, 116, 409–415.

[7] Bernardeschi, C., Domenici, A. and Masci, P. (2016) Modeling Communication Network Requirements for an Integrated Clinical Environment in the Prototype Verification System. In *The 2016 IEEE Symposium on Computers and Communication (ISCC)*, NW Washington, DC, USA, June 27–30, 2016, pp. 135–140. IEEE Computer Society, Messina, Italy.

[8] Bernardeschi, C., Domenici, A. and Masci, P. (2018) A PVS-Simulink integrated environment for model-based analysis of cyber-physical systems. *IEEE Trans. Softw. Eng.*, 44, 512–533.

[9] Bernardeschi, C., Domenici, A. and Saponara, S. (2019) Formal verification in the loop to enhance verification of safety-critical cyber-physical systems. *Electronic Communications of the EASST, Interactive Workshop on the Industrial Application of Verification and Testing, ETAPS 2019 Workshop*, 77, 1–9.

[10] Blochwitz, T.*et al.* (2011) The Functional Mockup Interface for Tool independent Exchange of Simulation Models. In *Proc. of the 8th Int. Modelica Conf.*, Linköping, Sweden, March 20th–22nd, 2011, pp. 105–114. Linköping University Electronic Press, Dresden, Germany.

[11] Bohrer, B., Rahli, V., Vukotic, I., Völp, M. and Platzer, A. (2017) Formally Verified Differential Dynamic Logic. In *Proc. of the 6th ACM SIGPLAN Conf. on Certified Programs and Proofs, CPP 2017*, New York, NY, USA, January 16–17, 2017, pp. 208–221. ACM, Paris, France.

[12] Bullock, D., Johnson, B., Wells, R.B., Kyte, M. and Li, Z. (2004) Hardware-in-the-loop simulation. *Transport. Res. Part C Emerg. Technol.*, 12, 73–89.

[13] Chen, X., Tang, J. and Lao, S. (2020) Review of unmanned aerial vehicle swarm communication architectures and routing protocols. *Appl. Sci.*, 10, 1–23.

[14] Cortés, J., Martínez, S., Karatas, T. and Bullo, F. (April 2004) Coverage control for mobile sensing networks. *IEEE Trans. Robotics Automation*, 20, 243–255.

[15] Cremona, F., Lohstroh, M., Broman, D., Lee, E.A., Masin, M. and Tripakis, S. (Nov 2019) Hybrid co-simulation: it's about time. *Softw. Syst. Model.*, 18, 1655–1679.

[16] Domenici, A. and Bernardeschi, C. (2021) A Logic Theory Pattern for Linearized Control Systems. In *The 6th Workshop on Formal Integrated Development Environment (F-IDE 2021) – Affiliated to NASA Formal Methods 2021, Virtual event, Electronic Proceedings in Theoretical Computer Science (EPTCS)*, May 24-25, 2021, in press.

[17] Domenici, A., Fagiolini, A. and Palmieri, M. (2018) Integrated Simulation and Formal Verification of a Simple Autonomous Vehicle. In *Software Engineering and Formal Methods (SEFM 2017), volume 10729 of LNCS*, Trento, Italy, September 4–5, 2017, pp. 300–314. Springer, Cham.

[18] Fax, J.A. and Murray, R.M. (Sept 2004) Information flow and cooperative control of vehicle formations. *IEEE Trans. Automat. Contr.*, 49, 1465–1476.

[19] Fitzgerald, J.S., Larsen, P.G. and Verhoef, M. (2007) *Vienna Development Method*, pp. 1–11. John Wiley & Sons, Inc, Hoboken, NJ, USA.

[20] Franchetti, F.*et al.* (April 2017) High-assurance SPIRAL: end-to-end guarantees for robot and car control. *IEEE Contr. Syst.*, 37, 82–103.

[21] Gasparri, A. and Oliva, G. (2012) Fuzzy Opinion Dynamics. In *The 2012 American Control Conference (ACC)*, NW Washington, DC, USA, June 27–29, 2012, pp. 5640–5645. IEEE Computer Society, Montréal, Canada.

[22] Gomes, C., Legat, B., Jungers, R.M. and Vangheluwe, H. (2017) Stable Adaptive Co-simulation: A Switched Systems Approach. In *IUTAM Symposium on Co-Simulation and Solver Coupling, number 35 in IUTAM Bookseries*, September 18–20, 2017, pp. 81–97. Springer, Cham, Darmstadt, Germany.

[23] Gomes, C., Thule, C., Broman, D., Larsen, P.G. and Vangheluwe, H. (May 2018) Co-simulation: a survey. *ACM Comput. Surv*, 51, 49:1–49:33.

[24] Henzinger, T.A. (1996) The Theory of Hybrid Automata. In *Proc. of the 11th Annual IEEE Symposium on Logic in Com-*

puter Science, NW Washington, DC, USA, July 27–30, 1996, pp. 278–292. IEEE Computer Society, New Brunswick, NJ, USA.

[25] Holt, J. and Perry, S. (2008) *SysML for Systems Engineering*. Institution of Engineering and Technology, Stevenage, UK.

[26] Jadbabaie, A., Lin, J. and Morse, A.S. (June 2003) Coordination of groups of mobile autonomous agents using nearest neighbor rules. *IEEE Trans. Automat. Control*, 48, 988–1001.

[27] Jalali, L., Mehrotra, S. and Venkatasubramanian, N. (Nov 2014) Simulation integration: Using multidatabase systems concepts. *Simulation*, 90, 1268–1289.

[28] Kar, S., Moura, J.M.F. and Ramanan, K. (June 2012) Distributed parameter estimation in sensor networks: nonlinear observation models and imperfect communication. *IEEE Trans. Inf. Theory*, 58, 3575–3605.

[29] Koenig, N. and Howard, A. (2004) Design and Use Paradigms for Gazebo, An Open-Source Multi-robot Simulator. In *The 2004 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, NW Washington, DC, USA, 28 September – 2 October, 2004, pp. 2149–2154. IEEE Computer Society, Sendai, Japan.

[30] Larsen, P.G.*et al.* (2016) Integrated Tool Chain for Model-Based Design of Cyber-Physical Systems: The INTO-CPS Project. In *The 2nd Int. Workshop on Modelling, Analysis, and Control of Complex CPS (CPS Data)*, NW Washington, DC, USA, April 11, 2016, pp. 1–6. IEEE Computer Society, Vienna, Austria.

[31] Larsen, P.G., Gamble, C., Pierce, K., Ribeiro, A. and Lausdahl, K. (2014) *Support for Co-modelling and Co-simulation: The Crescendo Tool*, pp. 97–114. Springer, Berlin Heidelberg.

[32] Leivant, D. (1994) Higher order logic. In Gabbay, D.M., Hogger, C.J., Robinson, J.A. (eds) *Handbook of Logic in Artificial Intelligence and Logic Programming*, pp. 229–321. Oxford University Press, Oxford, UK.

[33] Mahony, R., Kumar, V. and Corke, P. (2012) Multirotor aerial vehicles: Modeling, estimation, and control of quadrotor. *IEEE Robot. Autom. Mag.*, 19, 20–32.

[34] Manna, Z. and Pnueli, A. (1995) *The Temporal Logic of Reactive Systems: Safety*. Springer, New York.

[35] Masci, P., Curzon, P., Harrison, M., Ayoub, A., Lee, I. and Thimbleby, H. (2013) Verification of Interactive Software for Medical Devices: PCA Infusion Pumps and FDA Regulation as an Example. In *The EICS2013, 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, New York, NY, USA, June 24–27, 2013, pp. 81–90. ACM, London, UK.

[36] Muñoz, C., Narkawicz, A., Hagen, G., Upchurch, J., Dutle, A. and Consiglio, M. (Sept 2015) DAIDALUS: Detect and Avoid Alerting Logic for Unmanned Systems. In *Proc. of the 34th Digital Avionics Systems Conf. (DASC 2015)*, NW Washington, DC, USA, Sept. 13–17. 2015. IEEE Computer Society, Prague, Czech Republic.

[37] Narkawicz, A., Munoz, C., and Dutle, A. (2018) The Minerva Software Development Process. In Natarajan Shankar and Bruno Dutertre (eds.), *Automated Formal Methods*, volume 5 of Kalpa Publications in Computing, May 19–20, 2017, pp. 93–108. Moffett, CA, USA. EasyChair.

[38] Newell, J., Pang, L., Tremaine, D., Wassyng, A. and Lawford, M. (Jan 2018) Translation of IEC 61131-3 function block diagrams to PVS for formal verification with real-time nuclear application. *J. Autom. Reason.*, 60, 63–84.

[39] NIA 2003–03, NASA/CR-2003-212418 (2003) Rapid prototyping in PVS. Technical report. National Institute of Aerospace, Hampton, VA, USA.

[40] Oladimeji, P., Masci, P., Curzon, P. and Thimbleby, H. (2013) PVSio-web: a tool for rapid prototyping device user interfaces in PVS. *Electron. Commun. EASST*, 69, 1–8.

[41] Olfati-Saber, R., Fax, J.A. and Murray, R.M. (Jan 2007) Consensus and Cooperation in Networked Multi-agent Systems. *Proc. of the IEEE*, 95, 215–233.

[42] Owre, S., Rajan, S., Rushby, J., Shankar, N. and Srivas, M. (1996) PVS: Combining Specification, Proof Checking, and Model Checking. In Alur, R., Henzinger, T.A. (eds), *Computer-Aided Verification, CAV '96, number 1102 in LNCS*, pp. 411–414. Springer, Berlin, Heidelberg.

[43] Palmieri, M., Bernardeschi, C. and Masci, P. (2018) Co-simulation of semi-autonomous systems: the Line Follower Robot case study. In *Software Engineering and Formal Methods (SEFM 2017)*, volume 10729 of LNCS, Trento, Italy, September 4–8, 2017, pp. 423–437. Springer, Cham.

[44] Palmieri, M., Bernardeschi, C. and Masci, P. (2020) A framework for FMI-based co-simulation of human–machine interfaces. *Softw. Syst. Model.*, 19, 601–623.

[45] Platzer, A. and Quesel, J.-D. (2008) KeYmaera: A Hybrid Theorem Prover for Hybrid Systems (System Description). In Armando, A., Baumgartner, P., Dowek, G. (eds), *Automated Reasoning*, volume 5195 of LNCS, pp. 171–178. Springer, Berlin Heidelberg.

[46] Püschel, M.*et al.* (Feb 2005) SPIRAL: code generation for DSP transforms. *Proc. of the IEEE*, 93, 232–275.

[47] Wei, R., Beard, R.W. and Atkins, E.M. (2005) A Survey of Consensus Problems in Multi-agent Coordination. In *Proc. of the 2005 American Control Conf.*, NW Washington, DC, USA, June 8–10, 2005, pp. 1859–1864. IEEE Computer Society, Portland, OR, USA.

[48] Sander, I. and Jantsch, A. (Jan 2004) System modeling and transformational design refinement in ForSyDe. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 23, 17–32.

[49] Selic, B. (Sept 2003) The pragmatics of model-driven development. *IEEE Software*, 20, 19–25.

[50] Skorobogatov, G., Barrado, C. and Salami, E. (2020) Multiple UAV systems: a survey. *Unmanned Systems*, 8, 149–169.

[51] Smullyan, R.M. (1995) *First-Order Logic*. Dover Publications, Mineola, NY, USA.

[52] Wang, B. and Baras, J.S. (Oct 2013) HybridSim: A Modeling and Co-simulation Toolchain for Cyber-Physical Systems. In *The 2013 IEEE/ACM 17th Int. Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, NW Washington, DC, USA, October 30–November 1, 2013, pp. 33–40. IEEE Computer Society, Delft, Netherlands.