CAREER: Verifiable Execution and Simulation of Cyber-Physical Systems

Proposal

Jean-Baptiste Jeannin
Department of Aerospace Engineering
The University of Michigan (UM)
Ann Arbor, MI 48109-2140
Tel: (734) 764-6792
E-mail: jeannin@umich.edu

Suggested reviewers: Ranjit Jhala (UCSD), Sayan Mitra (UIUC), Sriram Sankaranarayanan (CU Boulder), Sanjit Seshia (UC Berkeley), Cesare Tinelli (U Iowa)

submitted to:
Software and Hardware Foundations (SHF)
Division of Computing and Communication Foundations (CCF)
Directorate for Computer and Information Science and Engineering (CISE)
National Science Foundation
Arlington, VA 22230
Tel: (703) 292-8950
Deadline: July 27, 2022.
Program Website: https://www.nsf.gov/pubs/2022/nsf22586/nsf22586.htm

CAREER: Verifiable Execution and Simulation of Cyber-Physical Systems

Overview

Modern cyber-physical systems such as cars and aircraft are responsible for expensive equipment and human lives. To ensure that they operate in a safe manner, it is therefore important to *formally verify* the underlying code. However today, tools for execution, simulation and verification of cyber-physical systems are largely disconnected, and formal verification often ends up verifying a model of the code rather than the code that is actually executed, creating a gap in the verification. The proposed research will combine techniques for execution, simulation and verification in one programming language, bridging this gap and offering a methodology for end-to-end verification of cyber-physical systems.

Intellectual Merit

We broadly identify three families of work: synchronous languages such as Lustre, Esterel and Signal focus on software that is high-level and *executable*; Simulink, Modelica, Zelus, Ptolemy and others focus on *simulation*; Differential Dynamic Logic and SpaceEx, as well as many other model-checking-based approaches, focus on *verification*. However having different tools and languages poses an issue: in practice, translating from the verified version of critical software in one tool to its executable or simulated version in another tool often requires a manual rewrite of the verified implementation – opening the door to human errors. In contrast, I believe that we need to *unify those approaches* to enable an *end-to-end verification* of cyber-physical systems without any manual translation. We propose the development of VExSim, a **new high-level programming language for the verification, execution and simulation of hybrid systems**. VExSim will be a *synchronous* language with *differential equations* and *dependent types*, and its verifier will generate verification conditions to be checked by external solvers. The language will compile down to the synchronous language Lustre, and this compiler will be formally verified, ensuring the absence of bugs in the compiler. This language will enable end-to-end verification of cyber-physical systems, be compatible with the SCADE industry standard, and compile to executable code in a formally verified way.

Broader Impacts

This program will educate and empower tomorrow's engineers to design cyber-physical systems with strong, end-to-end, formally verified guarantees. The PI maintains strong ties with industry, including with Toyota and Collins Aerospace. These connections will allow him to evaluate the new language on relevant projects provided by industry. The PI is planning to collaborate with a local middle school to enhance their curriculum and make the students more aware of safety issues and the availability of techniques to mitigate those issues, such as verification. The PI has also created a new class "AERO 350: Fundamentals of Aerospace Computing", to give undergraduates a a firm understanding of the use of computing in aerospace engineering. The class is central to the evolution of the teaching philosophy of the department to embrace the computer revolution of aerospace engineering, and it has recently been voted by the faculty as mandatory for all undergraduates in aerospace engineering starting in the Fall of 2020. The PI has also been teaching a graduate class "AERO 552: Aerospace Information Systems", and is revamping it to introduce a formal verification module, and to make it a sequence with AERO 350. In both courses, the PI will introduce the students to the programming language developed in this research program, giving them the elements and methodologies to perform end-to-end formal verification of software in their future occupations. Finally, the PI is active in the organization of the annual Verification Mentoring Workshop, having organized the 2020 edition and planning to organize the 2021 edition.

CAREER: Verifiable Execution and Simulation of Cyber-Physical Systems

1 Motivation and Overview

The presence of software is steadily growing in our everyday life, and computer programs are playing increasingly important roles in many aspects of our existence. In many cases, such as in the case of software running in trains, car, aircraft and spacecraft, but also in chemical plants or on thermostats, the software interacts directly with its physical environment, thus forming a *cyber-physical sytem* (CPS). Many cyber-physical systems, especially control software in transportation, hold responsibility for expensive equipment or human lives, and malfunctions could lead to catastrophic outcomes, including deaths of users or operators. Moreover, because of their physical nature, CPSs are often impossible to test extensively, especially with respect to rare corner cases. As a result, testing is not sufficient to attain reasonable levels of trust.

To achieve a high level of confidence in the design of CPSs, many researchers have advocated for the use of *formal verification*, which uses mathematical techniques to ensure safety (or liveness) with a very high level of confidence. However, the state of the art shows a divide between languages and tools that are used for executing CPSs, simulating CPSs, and verifying CPSs. In particular, many tools support execution and simulation with no or very limited support for formal verification; and many tools designed for verification will not enable direct modular design and execution of the verified code. As a result, many verification efforts are made on a *model* of the software (the cyber part of the CPS) rather than on the source code – or the binary code – that actually runs on the CPS. Verifying a model creates a trust gap where the engineer or researcher has to trust that the model of the code is faithful to the running code, which may not be true because of errors in creating the model, or even simply because of bugs in the compiler.

The current state of the art can broadly be classified in three categories: synchronous languages, such as Lustre, SCADE, Esterel and Signal, are languages that are high-level and *executable*; Ptolemy, Simulink, Modelica, Zélus and others focus on *simulation*; Differential Dynamic Logic and SpaceEx, as well as many model-checking-based approaches, focus on *verification*. This CAREER project will design a new programming language called MARVeLus that will combine the three approaches to provide **a unified approach to executing, simulating and verifying cyber-physical systems. Combining all three approaches will, for the first time, enable *end-to-end verification* of cyber-physical systems, where the only trusted components are the verifier and the model of the physical environment.**

We will model CPSs as *hybrid systems*, systems with both discrete and continuous behaviors: in a hybrid system, software typically affects the system in *discrete* ways – variable assignment, actuator update – while the environment is most often governed by *continuous* laws of physics. At a high-level, MARVeLus will be a *synchronous* language with *differential equations* and *s*: a synchronous approach inherits from decades of rigorous research on synchronous languages, and enables some compatibility with the SCADE suite, widely used to program CPSs; differential equations are needed to model the environment; and refinement types provide all the tools necessary to specify and verify CPSs.

Why is now the right time to do this work? Techniques for formal verification have made great progress over the last few years, and the field is maturing, especially in select application areas such as security or distributed systems []. For software not interacting with physics, there are now several tools performing verification of executable code [**?**, **?**], many of which use refinement types or refinement types. Automation has also improved dramatically, with verification of projects requiring less and less manual interaction, e.g., in the field of distributed systems. On the other hand, synchronous languages have existed for over 40 years and are now widely used in industry – Scade based on Lustre [], but also Esterel [] and Signal [], and verification techniques based on model-checking were developed early on []. However, this only accounts for the purely discrete case. The integration with continuous dynamics and differential equations [**?**] has been made quite recently at the language level – including use in industry in a new version of SCADE [] – but without verification capabilities for the

continuous part. Finally, verification of hybrid systems has also made much progress in recent years, in particular with the use of zonotopes and the automatic generation of invariants for continuous dynamics []. On all those fronts – formal verification in general and of hybrid systems in particular, integration of continuous dynamics with synchronous languages – many recent advances have been made. We believe the time is right to bring the verification of hybrid systems to the language level, to ensure the accurate verification of executable code.

Why build MARVeLus on a synchronous language? Synchronous languages such as Lustre, but also Esterel and Signal, were developed specifically to program embedded systems and cyber-physical systems in a safe way. For example, the Lustre language guarantees that any program will execute in bounded memory (with a known bound), and that its execution time will be predictable. The code generation is efficient, the result of decades of optimization development that we can leverage. Last but not least, Lustre is the basis of the SCADE tool, which is widely used in the aerospace and automobile industry, making it much more likely that MARVeLus will be used in industry on SCADE code.

Another tool widely used in some industries is Matlab/Simulink. However, Simulink is closed-source and does not have a widely accepted formal semantics, although there exist some academic attempts [**?**]. This makes it very difficult to develop sound proof techniques, since soundness needs to be proved with respect to a semantics. On the other hand, SCADE and Simulink are very close in spirit, and we expect many of our results to be applicable to the Simulink ecosystem in the long run (but, likely, without soundness claims).

Applications. To ensure its usability and adoption, the development of MARVeLus will be evaluated on some *case studies* and *collaborations with industry*. Our research agenda will include two case studies. The first case study will be on the Next-Generation Airborne Collision Avoidance System ACAS X, an industrial aircraft collision avoidance system currently under development by the Federal Aviation Administration. I have accomplished formal verification on ACAS X in the past, and am therefore very familiar with its design and specificities. The second case study will be the end-to-end verification of braking and swerving maneuvers for automobiles, for which I already collaborate with the Toyota Research Institute. This second case study is part of a broader vision of Toyota to equip their car with some formally verified modules ensuring the safety of the car at every instant. This application will likely lead to interesting technology transfer with Toyota, as well as their possible adoption of MARVeLus internally. The two case studies are chosen because they are industrial cases where reliability is extremely important. In some cases, similar case studies have already been achieved with other tools; but they usually only focus on verification, simulation *or* execution, whereas we will tackle all three aspects jointly. Applications and case studies are very important as they allow us to evaluate MARVeLus by identifying its strengths and weaknesses, thereby guiding its further development. During the development of MARVeLus, we hope to encounter new interesting case studies to further our experience with the language.

Outcomes and Impact. The research program outlined in this proposal will enable end-to-end formal verification of safety-critical applications such as aircraft collision systems. While current approaches to verification of cyber-physical systems typically verify a model of the software or the code at a high-level, we propose an approach that will lead to full verification of the cyber-physical system down to the binary code, leaving only two pieces to be trusted: the theorem prover and verifier performing the verification, and the dynamics of the environment. All other parts of the cyber-physical system will be mechanically verified, enabling the utmost level of trust and verification of safety-critical systems.

At a time where safety-critical, often autonomous, cyber-physical systems are responsible for expensive equipment and human lives, the systems' designers need powerful tools and methodologies to be able to guarantee that their creation is safe. This research program will design such tools, thereby empowering tomorrow's engineers to design cyber-physical systems with strong, end-to-end, formally verified guarantees.

Research plan. The specific research objectives are summarized below. More details can be found in Section 6.

1. **Design MARVeLus, a synchronous programming language with differential equations and refinement types.** Starting from the existing designs of the synchronous language Lustre [48], with differential

equation in the style of Zélus [21], we will design a refinement type system in the style of F* [106], enabling the verification of safety and liveness properties. We will also draw inspiration from the languages Ptolemy [96], Dafny [74], Agda [23] and Coq [16], among others. We will design this language in a rigorous way and equip it with a precise mathematical semantics (Task 1).

2. **Equip MARVeLus with verification and simulation tools.** Verification will be achieved by discharging proof obligations to external theorem provers such as Satisfiability Modulo Theories (SMT) solvers, in the style of F* [106] and Frama-C [29]. Simulation will leverage decades of work around Ptolemy [96] and Zélus [21] (Task 2).

3. **Apply MARVeLus on relevant applications of CPSs.** We will focus our attention on the industrial system ACAS X for aircraft collision avoidance, and emergency braking and swerving maneuvers for automobiles, in collaboration with the Toyota Research Institute (Task 4).

This research plan will be complemented by an education plan where our research will be integrated and directly used by students at the undergraduate and graduate levels. We will also collaborate with the engineering program at a local middle school to enhance the students' understanding of the safety requirements of the systems they will engineer.

2 Career Goals and Strengths of the PI

My background is unique in that I have experience doing research in **programming language design and theory, as well as formal verification of practical cyber-physical systems**, such as the industrial airborne collision avoidance system ACAS X. I am therefore in an ideal position to bridge the gap formed between simulation and compilation of cyber-physical systems, and their logic-based formal verification, as illustrated in Fig. 1.

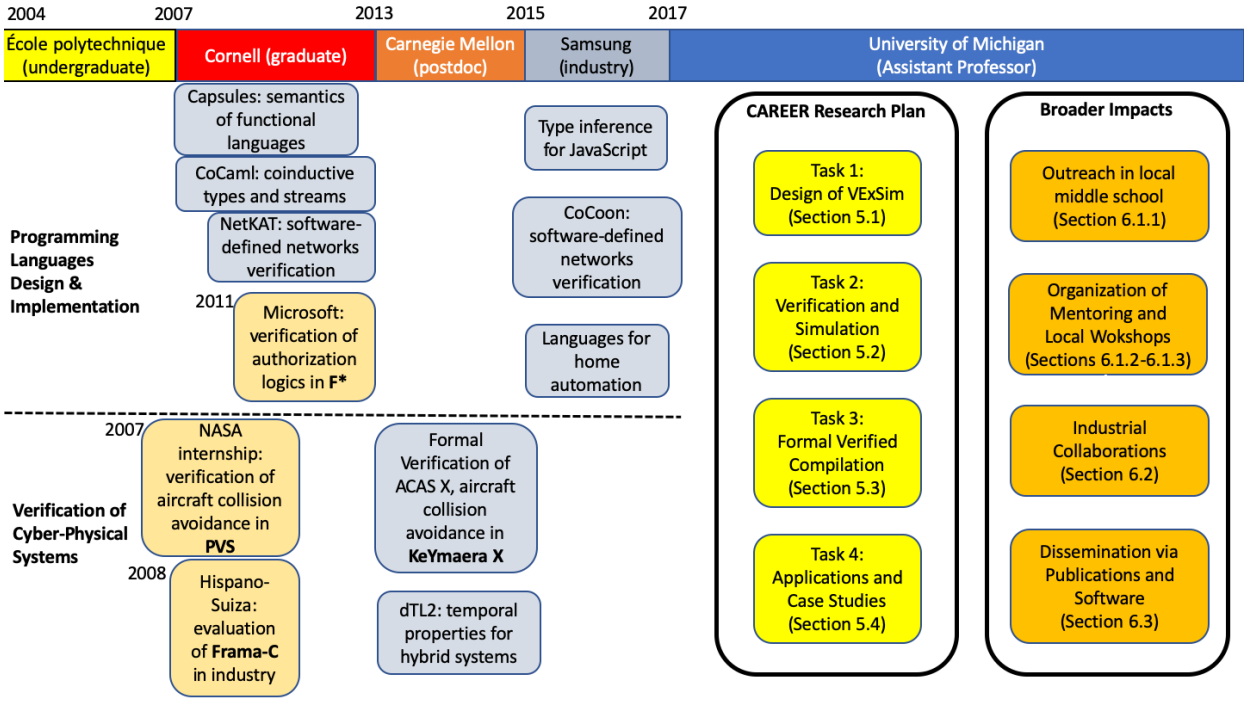

Figure 1: Overall career path and its relationship with the proposed work plan

More automatic hybrid systems proofs. With my PhD student Nishant Kheterpal and undergraduate Elanor Tang, I have developed an automated method for generating quantifier free formulations of obstacle avoidance that are efficiently machine-checkable [65]. Such formulations can be used in aerospace or ground robotics applications to

provably check whether an object on a planned trajectory will safely avoid an obstacle on its route. This method automatically transforms a quantified statement of obstacle avoidance, which cannot be tested efficiently or verifiably because of the universal (for-all) quantifier. This method leverages geometric intuition and is the first known method to automate the generation of quantifier-free safe (collision-free) regions in this way. We plan on extending our method from planar collision avoidance to consider more general questions of reachability in three or more dimensions.

Verification of numerical methods. With my PhD student Mohit Tekriwal, I have been working on formalizing the convergence properties of numerical programs. We have been working on developing an end-to-end verification framework for verifying the convergence of numerical solutions obtained by implementing a numerical algorithm in a programming language, which in this case is chosen to be C. There are two aspects to this framework:

1. **Verifying the mathematical properties of a numerical method**: This layer of proof is called the *property proof*. In this layer, we state a *functional model* for the algorithm we intend to implement. This functional model is a simple algorithm without intricate C semantics. This functional model is then proven correct with respect to a high level specification that is stated as a theorem statement in the Coq proof assistant.

   In this layer, the authors have first formalized the Lax–equivalence theorem to prove convergence of a family of finite difference schemes [108]. They have then built upon this work to prove the convergence of solutions obtained from iterative algorithms to the exact solution that has been proven to exist in the first work [109].

2. **Verifying that the functional model refines the exact implementation:** This layer of proof is called the *refinement proof*. In this layer, the functional model, which is developed in the previous layer, is proven to refine the actual implementation. This takes care of the programming error and the rounding errors introduced due to finite precision.

   In this layer, they are currently working on developing a floating point functional model for the iterative algorithms considered the second work. The floating point functional model will then be proven to refine the actual implementation of the algorithm. This development leverages the automation provided by VCFloat [99], which generates an annotated real expression from the floating point expression provided by the floating-point functional model, and VST [8], which provides a framework to prove refinement of the functional model to the actual implementation using its Floyd–Hoare logic automation.

By composing both the layers, the authors plan on developing an end-to-end verification framework. This layered approach is inspired by a work on C program verification of square root algorithm [9].

Language Design. I have extensive experience designing programming languages for stream programming, verification of software-defined networks and home automation. Those past designs have many similar challenges to the design of the MARVeLus language.

Software-Defined Networking. Beyond stream programming, I have experience designing languages and verification tools for software-defined networking. Software-defined networking is a recent trend in computer networking that enables central management and programming of computer networks, leading to better performance and a decrease in hard-to-track bugs. I am a co-designer of the NetKAT language [6], which provides a sound and complete axiomatization of a programming language for computer networks, based on Kleene Algebra with Tests (KAT) [70]. He is also a co-designer of the CoCoon language [101], which shows how to build correct by construction, software-defined networks.

Home automation. Finally, I have also co-designed the IoTa programming language for the automation of home tasks such as turning lights on and off or managing a thermostat [85]. IoTa comes with an internal checker that ensures that no two tasks can be contradictory or interact in undesirable ways, a common issue in home automation programming.

3 A motivating example: Formal Verification of Aircraft Collision Avoidance

ACAS X is the Next-Generation Airborne Collision Avoidance System, an industrial system destined to be installed on airliners as a successor of TCAS (Traffic Collision Avoidance System). Its development is funded by the

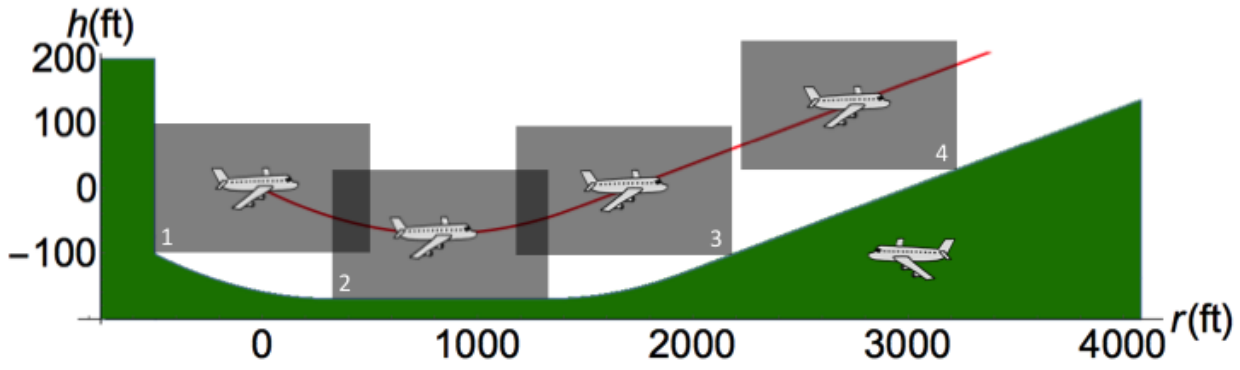

Figure 2: Nominal trajectory of the ownship (red) and safe region for the intruder (green), immediate response

Federal Aviation Administration as a response to a number of factors, including growing air traffic and the arrival of unmanned aerial systems (UAS). At the heart of ACAS X is a table whose domain describes the current state of an encounter, and whose range is a set of scores for each possible action [66, 67]. The table is obtained from a Markov Decision Process (MDP) approximating the dynamics of the system in a discretization of the state-space, and optimized using dynamic programming to maximize the expected value of events over all future paths for each action [66]. Near Collision events, for example, are associated with large negative values and issuing an advisory is associated with a small negative value.

A few years ago, I, along with some collaborators, performed a verification of ACAS X using the hybrid systems theorem prover KeYmaera X [57, 58, 56]. The work was successful and uncovered millions of individual bugs, some of which influenced later design decisions. However, after performing this verification, I realized that there was a discrepancy: the code that we verified consisted of a model (about 170 lines of code in KeYmaera X) along with an abstraction of the table (about 40 lines). But the code actually being executed consists of about 10,500 lines of Julia, along with a 994 MB. The abstraction of the table was proved correct with great confidence. But, on the other hand, the 10,500 lines of Julia were quite far from or 170-line KeYmaera X model. To me, this was very frustrating: the code that we have painstakingly verified was not actually the code running on the airplane, or even the code provided in the reference implementation designed by the Federal Aviation Administration for use by the avionics manufacturers. We have do better. We should be able to provide guarantees for cyber-physical systems, directly on the executable code. Hence, from the KeYmaera X verification was born the idea for the MARVeLus project.

Since ACAS X involves both *discrete* advisories to the pilot and *continuous* dynamics of aircraft, it is natural to formally verify it using hybrid systems [57, 58, 56]. However the complexity of ACAS X, which uses at its core a large lookup table—defining millions of interpolation regions—makes the direct use of hybrid systems verification techniques intractable. Our approach is different. It identifies *safe regions* in the state space of the system where we prove formally that a particular advisory, if followed, prevents all possible NMACs from any aircraft position and velocity in that safe region. Then it *compares* these regions to the configurations where the ACAS X table returns this same advisory. Moreover our safe regions are *symbolic* in their parameters, and can thus be easily adapted to new parameters or new versions of ACAS X.

Figure 2 depicts an example of a head-on encounter and its associated safe region for the advisory "Climb at 1,500 ft/min", projected in a vertical plane with both aircraft. It is plotted in a *coordinate system fixed to the intruder* and centered at the initial position of the ownship. The ownship, surrounded by the puck (grey safety region), starts at position 1 and traces out a trajectory following the red curve. It first accelerates vertically with $g/4$ until reaching the desired vertical velocity of $+1{,}500\text{ft}/\text{min}$ at position 3. It then continues to climb at $+1{,}500\text{ft}/\text{min}$. The green safe-region indicates starting points in the state space for which the aircraft will remain safe for the duration of the encounter when following the "Climb at 1,500 ft/min" advisory in any way. Note that no safe region exists above the trajectory since the ownship is allowed to accelerate vertically at greater than $g/4$ or climb more than $+1{,}500\text{ft}/\text{min}$.

Our results provide refinement characterizations of the ACAS X behavior to provide a clear and complete picture of its performance. Our method can be used by the ACAS X development team in two ways. It provides a mathematical proof – with respect to a precise hybrid systems model – that ACAS X is absolutely safe for some configurations of the aircraft. Additionally, when ACAS X is not safe, it is able to identify unsafe or unexpected behaviors and suggests ways of correcting them. Our approach of first formally deriving safe regions and then comparing them to the behavior of an industrial system is, as far as we are aware, the first of its kind in the formal verification of hybrid systems.

More recently, my group has applied similar techniques to formally verify horizontal (turning right and left) collision avoidance maneuvers for UAVs [3], as well as collision avoidance maneuvers for cars in collaboration with Toyota – both turning only [2] and braking-while-swerving maneuvers [1].

4 Intellectual Merit

Unification of approaches. An important intellectual contribution of this career project is to provide a unified language to bring approaches to verification, execution, and simulation together and enable communication between them. By doing so, the MARVeLus language will provide the necessary tools and methodologies to empower programmers and researchers to create end-to-end proofs of their cyber-physical systems.

Verification. We will design MARVeLus as a *synchronous* programming language with *differential equations* and *refinement types*. The two most important influences will be the F* programming language [106] and the Zélus programming language [21], itself based on the Lustre synchronous language [48]. To our knowledge this will be the first use of refinement types in a synchronous language dedicated to cyber-physical systems.

The verification will leverage existing verification techniques by identifying *proof obligations* in the program and proving them in existing tools such as SMT solvers [30, 17] or Mathematica [117]. Importantly, we will focus on proof automation, and strive to avoid manual proofs of obligations in an interactive theorem prover, because of their often prohibitive human effort.

Execution. To enable execution and simulation of the system, the language will have to be *deterministic*, contrarily to systems centered on verification like differential dynamic logic [91]. However, as demonstrated by the ACAS X project [58, 56, 68], verification is often easier on an abstracted, nondeterministic program. Central to the programming language will be the notion of *refinement, naturally enabled by refinement types*: verification can be achieved on a nondeterministic program that is then refined to a deterministic program used for execution and simulation. Such a refinement is only sound if the behaviors of the deterministic program are a subset of the behaviors of its nondeterministic counterpart.

One technical issue is determining what parts of the program to compile for execution. Indeed, for the execution on a physical system, the variables modeling the environment should not be compiled. A first thought is that discrete variables are part of the controller and should thus be compiled, whereas continuous variables are part of the environment and should not be compiled. However this is not true: there exist both continuous controllers – e.g., a PID controller – and discrete jumps in the environment – e.g., a ball bounce. We propose to solve this issue by introducing a simple type system, allowing the programmer to explicitly specify which variable is part of the environment and which is part of the controller. Interactions between those two types will be done via explicit sensors and actuators, similarly to [21, 31].

Simulation. In the simulation we will address issues like precision of the simulation of differential equations, and identification of zero-crossings, leveraging previous work done in Zélus [21] and Ptolemy [31]. This work on precision of the simulation will give us some intuition on the robustness of the verification and the compilation. I plan to use it to extend the verification with robustness guarantees, with respect to both floating-point arithmetic errors and perturbations in the environment.

5 Brief Overview of Related Research

5.1 Language and Tools for Cyber-Physical Systems

Synchronous languages such as Lustre/SCADE [48], Esterel [22] and Signal [73] are widely used in industry to create software for airplanes and cars, in part because they are capable of generating high assurance C code. However, based on discrete time, they do not allow the explicit modeling of a continuous environment. Nonetheless, they are a compelling success story with a wide use in industry, and an experience to build upon. Bourke et al. [20] formally verified a Lustre compiler in the Coq proof assistant. In general, cyber-physical systems are notoriously difficult to design correctly [104].

The Ptolemy and Ptolemy II projects [31, 96] leverage the principles of design by contract to cyber-physical systems. The focus is on modeling, simulation, and design of concurrent, real-time, embedded systems, with comparatively very little work on formal verification, apart from a few rare exceptions [27, 12] using model-checking [27] or the Real Time Maude tool [87, 12]. Simulink [88] and Modelica [40] can simulate complete systems, including a continuous environment, and Simulink offers code generation without correctness guarantees. The Zélus language [21] builds upon the Lustre synchronous language and adds ordinary differential equations to simulate a controller in its physical environment.

On the verification end of things, seminal works by Artemov et al. [10] and Alur et al. [5] created a basis to reason about hybrid systems. Model checking has been adapted to reasoning about hybrid sytems and hybrid automata [51]. Differential Dynamic Logic [91] extends Dynamic Logic [49] with differential equations to formally verify important safety and liveness properties of hybrid systems. SpaceEx [38] and CORA [4] explore reachability properties of hybrid systems with numerical approximations and zonotopes, enabling scalability. S-Taliro [7] and Breach [32] can find counter-example in Matlab/Simulink models. There are many approaches to generating invariants for cyber-physical systems, e.g., [44, 45, 103, 25, 26, 102, 100, 110, 47], and for synthesis of control algorithms [97, 98, 86].

Autocoding [63, 116, 115, 114] is an approach that compiles high-level code with formal proofs of good behavior, into low-level code. Although autocoding provides tools for verification and execution, it does not provide support for trustworthy simulation of the system. Other languages have recently been designed for different specialized purposes: the Paracosm [77] language focuses on testing of autonomous driving systems, the PGCD language [15] on robot programming and verification, and the Scenic language [39] on scenario specification and scene specification for testing machine learning algorithms, especially vision. HyST [13] is a source transormation tool for hybrid automaton models, but does not provide a high level of abstraction to build large systems. Wongpiromsarn et al. [118] provide new conditions for periodic hybrid automata. Closest to this career project, Bohrer et al. [19] present an end-to-end verification of models verified in Differential Dynamic Logic. However their source language based on hybrid programs is fairly low level, lacks modularity, such as basic support for declaring functions, and compositionality, such as parallel composition.

5.2 Languages for Formal Verification

There is an enormous amount of previous work on languages and techniques for formal verification of programs, and it is impossible to list them all here. We will focus our discussion on logic-based approaches stemming from variants of Hoare logic or refinement types, since these are the techniques used in this proposal. Although we do not review it in detail for lack of space, there is also an extensive amount of work on verifying hybrid systems using temporal logics [94, 95, 79] and model checking [28, 14, 52].

The seminal works of Floyd [37] and Hoare [54] on assigning meanings to programs has been extended into dynamic logic [36, 49], the μ-calculus [69], and Kleene algebra with tests [6, 70]. Modern tools such as Dafny [74], Frama-C [29] and Krakatoa [35] use variants of these techniques at their heart. Dependent types [**?**, 11, 119] is an alternate approach to program verification, and its modern incarnations include he F* project [106, **?**, 107], Dependent ML [**?**], Agda [23], and Zombie [24, 105]. Interactive theorem provers allow users to provide formal proofs of mathematical theorems, including proofs of programs. They include Coq [16], Isabelle [90], HOL and HOL Light [50], PVS [89], and others. Finally, the logic TLA [71] is specialized to reason about and formally

verify protocols involving distributed systems.

I am very familiar with a variety of those tools. I have worked in the past with the KeYmaera X theorem prover [41, 58, 58] during my postdoctoral studies, the F⋆ language [106, 55] at Microsoft Research, the PVS theorem prover [89, 53] at NASA, the Frama-C framework [29] in industry at Hispano-Suiza, as well as the Coq theorem prover [16] and dynamic logic in recent projects [**?**] (Fig. 1).

5.3 Aircraft Collision Avoidance

Many efforts have explored developing correct and comprehensive guarantees about collision avoidance decisions over a system's state space. Collision avoidance algorithms are developed for both horizontal and vertical motion in 3D in [82, 83] for polynomial trajectories with a finite time horizon, and formally verified with PVS. In [81], the logic for the Traffic Collision Avoidance System TCAS, which currently runs on commercial aircraft, is formalized in PVS and used to identify straight-line encounter geometries that generate advisories in a noiseless environment. The ACAS X system logic [66] is based on a policy that results from optimizing an Markov Decision Process (MDP) using value iteration to minimize a set of costs; [113, 72] analyze the state space of a similar MDP using probabilistic model checking and an adaptive Monte Carlo tree search respectively, to identify undesirable behavior.

Collision avoidance algorithms called KB3D and KB2D are formally verified using the PVS theorem prover in [34, 42], with a continuous state space in 2D and 3D, but assuming the velocity vector changes instantaneously according to an advisory, and then has no acceleration thereafter; realistic performance is evaluated for multiple aircraft using a low-fidelity simulation, and the proofs do not guarantee conflict avoidance. [84] uses a rigorous approach to construct a formally verified well-clear volume based on straight-line dynamics with no acceleration. Efforts that use a hybrid system model to develop safe horizontal maneuvers can be found in [43, 76, 93, 111].

The aforementioned results all verify a model of the considered system and of its software, and are thus unable to provide an end-to-end proof of correctness for collision avoidance. In contrast the MARVeLus language will enable such end-to-end proofs in the context of cyber-physical systems.

6 Research Plan

We will design MARVeLus, a *synchronous* programming language with *differential equations* and *refinement types*. The two most important influences will be the F⋆ programming language [106] and the Zélus programming language [21], itself based on the Lustre synchronous language [48]. Although F⋆ focuses on refinement types and Zélus on synchrony and differential equations, both languages are part of the ML family of languages [80, 75], and we are confident their designs can be integrated. Further influences, among others, will come from Dafny [74] and differential dynamic logic [91] for the verification part; Ptolemy [27] for the simulation part; and Esterel [22] and Signal [73] for the execution part. **The philosophy for MARVeLus can thus be summarized as "Zélus with refinement types", "Lustre with differential equations and refinement types", or "Synchronous F⋆ with differential equations".** We expose refinement types and the verification engine in Task 2.1.

Throughout the design of the language, we will be particularly attentive to retain some level of compatibility with the SCADE suite [18, 33] (itself based on Lustre). SCADE is based on synchronous languages, and is widely used as a development platform for cyber-physical systems, including by large companies such as Airbus Commercial Aircraft (designing commercial airliners Airbus 320, Airbus 350 and Airbus 380 among others) and Airbus Space (designing and operating the Ariane rocket and space launcher), who develop many of today's most safety-critical cyber-physical systems. This compatibility with the industry standard will be a key asset for industry adoption. Initially we will focus on ordinary differential equations with a certain degree of regularity (typically Lipschitz-continuity), ensuring existence, unicity and "niceness" of solutions. In the long run we will consider adding more complicated continuous dynamics such as differential algebraic equations.

MARVeLus is envisioned to be a realistic, practical, usable language with all the features expected of a modern programming language, e.g., constructs for abstraction and modularity. Even when not using refinement types, a strong typing discipline will be enforced to find bugs early on in the development process, and to ease the verification process, in the spirit of ML.

We will first define the discrete part of MARVeLus (Task 1), then its continuous part (Task 2), then focus on

Applications.

Task 1 – Synchronous Programming with Refinement Types

In this first task, we focus strictly on the *discrete* part of our language, and on the simulation and execution. During our preliminary work, we have realized that developing a refinement type system for synchronous programming is a major undertaking by itself, and that we would need to separate the design of discrete MARVeLus from incorporating differential equations in our language.

Task 1.1 – Discrete MARVeLus: Syntax, Semantics and Simple Type System

We will first define the *discrete part* of MARVeLus, without refinement types or verification capabilities. In our initial design for this discrete part (i.e., including streams but without differential equations), all expressions generate (infinite) streams of basic values (typically integers).

In MARVeLus, the syntax of expressions is inspired by the discrete fragment of Zélus, which is itself inspired from Lucid Synchrone, an ML version of Lustre:

$e ::= \; c$	Constants
$\mid \; x$	Variables
$\mid$ let $x : \tau = e$ in e	Let-binding
$\mid \; \lambda x.e$	Function
$\mid \; e \; x$	Application
$\mid (e_1,e_2) \mid$ fst $e \mid$ snd e	Pairs
$\mid$ if x then e_1 else e_2	Branching
$\mid$ let rec $x : \tau = e$ in e	Recursion
$\mid \; e : t$	Type Annotation
$\mid \; c$ fby e	Streams

Trace predicates $\phi,\psi ::= p$
$\mid \phi \vee \psi$
$\mid \phi \wedge \psi$
$\mid \neg \phi$
$\mid \phi \, \mathcal{U}_{[t,t']} \, \psi$

base types $b ::=$ unit $\mid$ float $\mid$ int $\mid$ bool

types $\tau ::= b \mid \tau\{\phi\} \mid \tau \times \tau \mid (b \times b)\{\phi\} \mid \tau \to \tau$

Most of the constructs, including λ-abstractions, applications, variables, (recursive) let-bindings and pair construction and destruction, are standard from ML. To this we add stream construction using the fby constructs of Lustre: given two streams $c = (c_1,c_2,c_3,...)$ and $e = (e_1,e_2,e_3,...)$, the stream c fby e has the same first element at c, and the rest of the stream is e delayed by one time step. The resulting stream is $(c_1,e_1,e_2,e_3,...)$.

Preliminary Work. Our initial design is based on existing synchronous languages, especially Lustre and Lucid Synchrone. We have started from these designs and iterated a few times to build a language amenable to verification.

Streams, i.e., infinite list of elements, are an essential part of the design of synchronous languages. I have explored how to program rational coinductive types, including regular streams, using recursive and corecursive methods in the CoCaml language [59, 60, 61]. In a nutshell, the idea is that coinductive types (such as streams) are very similar to inductive types (such as linked lists), and they ought to be programmable using some sort of recursive function. However there is a catch: using the standard semantics of the language, those (co)recursive functions would never finish on coinductive types. The CoCaml language creates constructs to overcome this difficulty, by modifying the semantics in a way that ensures termination. In practice, computing results of recursive functions applied to coinductive types involves computing a fixed point using a solver built into the language or provided by the programmer.

Task 1.2 – A specification system based on Signal Temporal Logic

In many applications, traditional input/output specifications (e.g., à la Floyd-Hoare Logic) are inadequate for cyber-physical systems. For example, in the case of airplane or car collision avoidance, we want to be able to express that "the vehicles dis not collide throughout the execution" rather than "the vehicles are properly separated at the end of the execution". This is even more true in the case of reasoning with streams as part of synchronous

programming. Since the value of a variable is really an infinite stream, representing a function from time to a scalar, expressing properties about variables requires an easy incorporation of time constraints in our specifications.

Research Question: What is the right specification language for cyber-physical systems described in a synchronous language?

The state-of-the-art in cyber-physical-systems verification typically uses Signal Temporal Logic, a variation of Metric Interval Temporal Logic. Signal Temporal Logic was initially designed for *monitoring* of signals [].

Although refinement type systems come in many flavors and are very flexible, they typically only express properties about the end state of a system, but do not say anything about its intermediate states, which can cause issues when verifying safety properties of cyber-physical systems. A similar phenomenon happens with the expressivity of differential dynamic logic with respect to temporal properties. In the past, I have developed the differential temporal dynamic logic dTL^2 [62] to express temporal properties of Linear Temporal Logic (LTL) [94] in the context of dynamic logic.

In this task, based on our experience with dTL^2, we will extend the refinement type system of MARVeLus by adapting our works of Task 2.1 to enable the expression of temporal properties (of LTL) in refinement types of MARVeLus. We will then take things further and allow our system to prove properties that can be expressed in Signal Temporal Logic [78], an extension of LTL. Signal Temporal Logic is similar to LTL but further includes timing information about events. This extension to STL was suggested to us by Toyota Research Institute, who uses STL extensively internally, and who recently funded one of my projects. We note however that properties based on STL are typically undecidable, so in some cases some manual proof effort from the programmer will be necessary.

When completed successfully, this task will enable temporal properties about programs to be formally proved in MARVeLus, greatly enhancing the expressive power of the language.

Preliminary Work. My previous experience with Differential Dynamic Logic and its application to ACAS X verification, has been that dynamic-logic-style postconditions are often not expressive enough for interesting properties in cyber-physical systems. As a result, I have designed two extensions of Differential Dynamic Logic. First, I have developed Differential Temporal Dynamic Logic with Nested Modalities dTL^2 [], which enables specifications in a significant subset of Linear Temporal Logic (LTL). Second, with my PhD student Hammad Ahmad, we have developed Signal Temporal Dynamic Logic (STdL), where specifications belong to a significant subset of Signal Temporal Logic (STL).

Initial Design of Discrete MARVeLus. The syntax of refinement types is modeled on F* and LiquidHaskell, with some simplifications for conciseness: `x:int{x>0}`describes the variable `x`, which is specified to be a positive integer. Our refinement type system is inspired by Liquid Haskell [112], augmented with pairs. We make the following extensions. Most importantly, we extend the theory to include typing rules for streams, and trace predicates for expressing Linear Temporal Logic (LTL) properties. Indeed, interesting predicates on streams can often be expressed using LTL formulas. In the future, we envision extending those properties with fragments of Signal Temporal Logic (STL) or maybe CTL, depending on the needs of case studies. We add refinement types as an annotation to base types, allowing the user to supply a logical quantifier on the type, in the form of a boolean proposition ϕ that may depend on values present in the system. Any type specifications t without refinements are implicitly converted to the form $t:\{\,\mathsf{true}\}$

Synchronous languages allow us to express variables as streams, which essentially allows us to reason about how program states change over time. This evolution of a program over time can be thought of as a trace of a program, which is recursively constructed as a current value c concatenated (fby) with a formulation to derive the next value e, such that $e' = c$ fby e. Taking advantage of this structure, we are able to express system properties as Linear Temporal Logic (LTL) formulae, which lift normal Boolean predicates into the domain of streams, by expressing when or if such predicates are satisfied at given points along the trace. For instance, $\Box p$ is satisfied if the predicate p is satisfied for all instants of the trace, that is, the current value satisfies p and the stream of future values satisfies $\Box p$.

Task 1.3 – A Refinement Type System for Discrete MARVeLus

Research Question: How to incorporate refinement types with a synchronous language?

The type system will based upon that of standard Zélus, with the addition of a separate refinement-type checker. This enables us to leverage the existing type system to check base types and further refine it with our new typing rules. A simple example of of a typing rule relates the fby expression operator with the $\Box$ temporal expression operator:

$$\frac{\Gamma \vdash c : b\{p\} \qquad \Gamma \vdash e : b\{\Box p\}}{\Gamma \vdash c \text{ fby} e : b\{\Box p\}} \text{ (ALWAYS)}$$

Intuitively, this rule says that is refinement p is true in both the initial state of c and everywhere in e, then it should be true everywhere in the stream c fby e.

Although simulation can serve as an approximation for system behavior, embedded systems will ultimately need to interact with the real world. We designed the language to make switching between simulation and real-world execution as seamless as possible, requiring no code changes to do so. We add an additional annotation on variables exclusively defined in the main function of the program to specify its connection to real-world sensors and actuators. These serve no purpose in the verification process, and will allow the compiler to link MARVeLus variables to their physical sensor and actuator counterparts.

Completeness. A desirable, foundational result of our refinement type system is its completeness, which we will prove. We expect the completeness proof to use proof techniques coming from type theory, but also exploiting ideas from completeness results of axiomatizations of temporal logics.

Preliminary Work. We have started developing a refinement type system for a stream calculus. Most constructs allow for relatively straightforward proof rules, but certain cases are more difficult.

For example, the rules for the if construct are somewhat trickier than we initially expected. A possible typing rule for an if statement is the following []:

$$\frac{y \text{ fresh} \qquad \Gamma \vdash x : \mathsf{bool} \qquad \Gamma, y : \{\mathsf{int} : x\} \vdash e_1 : \tau\{\Box\phi\} \qquad \Gamma, y : \{\mathsf{int} : \neg x\} \vdash e_2 : \tau\{\Box\phi\}}{\Gamma \vdash \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau\{\Box\phi\}}$$

If we want to prove that a property ϕ is true throughout an execution (i.e.,), a possible similar rule for streams could be built as:

$$\frac{y \text{ fresh} \qquad \Gamma \vdash x : \mathsf{bool\ stream} \qquad \Gamma, y : \{\mathsf{intstream} : x\} \vdash e_1 : \tau\{\Box\phi\}\ \mathsf{stream} \qquad \Gamma, y : \{\mathsf{intstream} : \neg x\} \vdash e_2 : \tau\{\Box\phi\}\ \mathsf{stream}}{\Gamma \vdash \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau\{\Box\phi\}\ \mathsf{stream}}$$

However, the premises of this rule are much too strong: the rule essentially says that if the condition (x, a stream of booleans) is identically true for all time, then we can conclude as to the type of the if-expression; and if the condition is identically true for all time, then we can similarly conclude. But the rule does not allow to conclude anything in the cases where the condition x is true for some time steps and false for others, which is unsatisfying. Essentially, we have a quantifier order issue: the rule says that something of the form $\forall t.A$ implies $\forall t.B$, but what we really want is the much stronger $\forall t$, A implies B. Instead, we want to strengthen the rule to allows us to conclude for each time step independently of the others. Such a typing rule would have a form similar to:

$$\frac{y \text{ fresh} \qquad \Gamma \vdash x : \mathsf{bool\ stream} \qquad \Gamma, y : \{\mathsf{intstream} : x\} \vdash e_1 : \tau\{\Box(x \Rightarrow \phi)\}\ \mathsf{stream} \qquad \Gamma, y : \{\mathsf{intstream} : \neg x\} \vdash e_2 : \tau\{\Box(x \Rightarrow \phi)}{\Gamma \vdash \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau\{\Box(x \Rightarrow \phi)\}\ \mathsf{stream}}$$

In this new rule, because the condition x appears under the time quantification (embodied by the $\Box$ construst), the time quantification is much stronger and as desired.

Task 1.4 – Implementation

My students and I will implement the features described in the three previous tasks, and we will build a prototype of our synchronous language with refinement types The implementation will be in OCaml as my group has extensive experience with it, and to leverage some features of the existing Zélus compiler, on which our initial implementation will be based. In the long run, my plan is to integrate our development with the Zélus compiler, as this compiler already has significant industry traction, in particular among the Ansys company that commercializes SCADE, and its customers. By integrating our verification capabilities in an existing tool, we hope to encourage industry adoption and to minimize friction in using our tools.

Preliminary Work. My group and I have started implementing verification capabilities in an initial design of MARVeLus. Our initial implementation is based on an extension of the Zélus compiler, only exploiting its discrete subset. We add a compiler pass in the Zélus compiler, just after basic type-checking. This pass checks refinement types and generates proof obligations that are then sent to the Z3 SMT solver. Our current implementation can currently prove simple examples, such as proving that, under adequate assumptions, a discrete version of a thermostat. One graduate student and four undergraduate students are currently working on the project.

Task 2 – Incorporating Continuous Differential Equations

Once we have a working synchronous language with refinement types and temporal specifications, we will add support for incorporating continuous dynamics in the form of ordinary differential equations (ODEs), along with refinement type rules enabling proofs of interesting properties on those ODEs.

Task 2.1 – A Refinement Type System for Differential Equations

Research Question: How to incorporate reasoning about differential equations, including continuous invariants, in a refinement type system?

Declaring variables following Ordinary Differential Equations. Differential equations will be specified directly in the program, All differentiations will be made with respect to time, and we will not consider partial differential equations. Introducing an ordinary differential equation will be similar to introducing a recursive variable or stream. For example, to introduce a real variable x following the differential equation $x' = 4x + y^2$ with initial condition $x(0) = 2$, the programmer will be able to write the declaration

```
let der x = 4 * x + y * y init 2
```

where this construct is inspired by Zélus, and the `der` keyword refers to the first derivative.

Formally, we extend the set of expressions to incorporate declarations of differential equations:

$$e ::= \ldots \mid \texttt{der}\ x = e\ \texttt{init}\ e$$

Zero-crossings The interface between the discrete and continuous parts of MARVeLus will be achieved using zero-crossings, inspired from the Zélus language. We will develop specific refinement typing rules to handle zero-crossings.

Reasoning about ODEs. We These rules will be inspired by some of the differential equations proof rules of differential dynamic logic. The MARVeLus language will be limited to Ordinary Differential Equations with time derivatives.

For example, give the definition of x above, we would like to be able to prove that the variable x is always positive, which come down to proving that $x \geq 0$ is an invariant of the differential equation. Under the assumption that $x \geq 0$, we can prove that the derivative of x is also positive, and using a simple invariant rule, that x is always positive.

Completeness. Along with developing a Refinement Type System for the Differential Equation part of MARVeLus, we will extend our completeness of discrete MARVeLus to take into account the continuous dynamics. I expect this proof to follow similar arguments as the proofs of completeness of differential dynamic logic d$\mathcal{L}$, and its temporal counterparts dT$\mathcal{L}^2$ and STd$\mathcal{L}$, especially the complete proof theory of d$\mathcal{L}$ [92].

Task 2.2 – Different Compilation Modes for Simulation and Execution

Our compiler will allow for both simulation and execution of cyber-physical systems, using different compilation modes. However, in the case of execution, only certain parts of the program should be compiled, namely the parts that are not part of the environment. This leads to the following research question:

> **Research Question:** How does the programmer specify and the compiler identify which parts of the program to compile for execution, and which should only be run for a simulation?

Program variables and environment variables. Every variable of the language can be: (i) either discrete or continuous; and (ii) executed by the system or effects of the environment. Some variables might take both discrete and continuous transitions. Naively, one could think that discrete transitions are always executed by the system (software is typically discrete), while continuous transitions are effects of the environment's mechanical and physical laws. While this is typical, it is not always the case: proportional-integral controllers are continuous (even though often implemented in discrete software), while ball bounces can be modeled as discrete events executed by the environment. Each variable's type will reflect those characteristics, and the type system of the language will be designed accordingly.

Determinism and Nondeterminism. Closely related to the dichotomy between program and environment variables, lies the tension between determinism and nondeterminism. On the one hand, to be compiled and executed on hardware, a program needs to be deterministic, and the variables it uses need to have specific initial values. On the other hand, there is often uncertainty on environment variables, and Finally, to simulate a nondeterministic program or an uncertain variable, a value must be chosen, typically sampled from a certain probability distribution. The default will be a uniform distribution but we will allow the programmer to specify different distributions for simulation purposes.

Real numbers and floating-point numbers. Physical variables in a physical environment should typically be represented as real numbers, while program variables representing real numbers are typically represented and compiled as floating-point numbers. For this reason, we distinguish real numbers of type real, from floating-point numbers of type float. (Of course floating-point numbers also come in different levels of precision, half-precision, single-precision, double-precision, etc.) The type system will ensure that real variables cannot be program variables that would be compiled.

Even though operations on floating-point numbers can sometimes behave in unexpected ways (diverging approximations, underflows, overflows, etc.), it is often easier to carry out formal verification In MARVeLus, we will include a compilation flag for the refinement-type-checking pass, which will consider floating-point numbers as reals for verification purposes. Based on our experience on verifying ACAS X [] but also on verification of numerical schemes [], it is important to give the ability to the programmer to make the simplifying assumption that operations made on floating-point numbers behave as if they were made in the field of real numbers. Many verification conditions become simplified (e.g., overflows do not have to be dealt with), making the verification task more tractable. Of course, this simplification comes at a cost: if an unexpected event such as an overflow happens, then the guarantees of the verification are not valid anymore.

A similar distinction exists between (mathematical) natural numbers and machine integers. Beyond the trivial difference that natural numbers are only positive while machine integers can be negative, mathematical natural numbers cannot overflow, and are e.g. represented by the type nat in Coq. In MARVeLus, we give the type nat to natural numbers and the type int to integers to mark this distinction, and we will also include a flag to conflate them for verification as needed by the programmer.

Preliminary Work. Although we do not have preliminary work directly related to this task yet, in the past I have worked with collaborators on Dynamic Code Splitting for Javascript [46]. In that work, we allow the programmer to write a web application by writing just one program in Javascript, and our compiler automatically partitions the executable code in a client side and server side, adding explicit communication when needed. We expect the kind of code analysis of our Javascript work to be relevant in determining which part of the program to compile to hardware (for execution), and which part to simulate (for simulation).

Task 2.3 – Formally Verified Simulation

Simulation of dynamical systems that use differential equations For example, the Zélus tool uses off-the-shelf On the other hand, they

Use to enhance verification. Of course, we do recognize that a simulation can only be as good as its model. Nonetheless, the use of the unverified solver `ode45` by the Zélus team has led to the discovery [**?**], and I believe there is tremendous value in having a formally verified simulation, with assured and verified error bounds.

Finally, we will take advantage on past and ongoing work by Bourke et al. on formal verification of Lustre compilation []. While we do not plan to directly

Preliminary Work. We have already formalized certain results pertaining

Task 2.4 – Implementation

We will extend discrete MARVeLus to handle differential equations following the extended refinement type system.

Task 3 – Applications and Case studies

To guide the development of MARVeLus and ensure that the design decisions are adequate, we will use the language on three different kinds of applications: a ground robot and a quadcopter in the lab; industrial applications for ACAS X and car collision avoidance; and as a homework in a graduate class with a verification module.

Task 3.1: Small ground robot: the M-Bot Mini Ground Robot

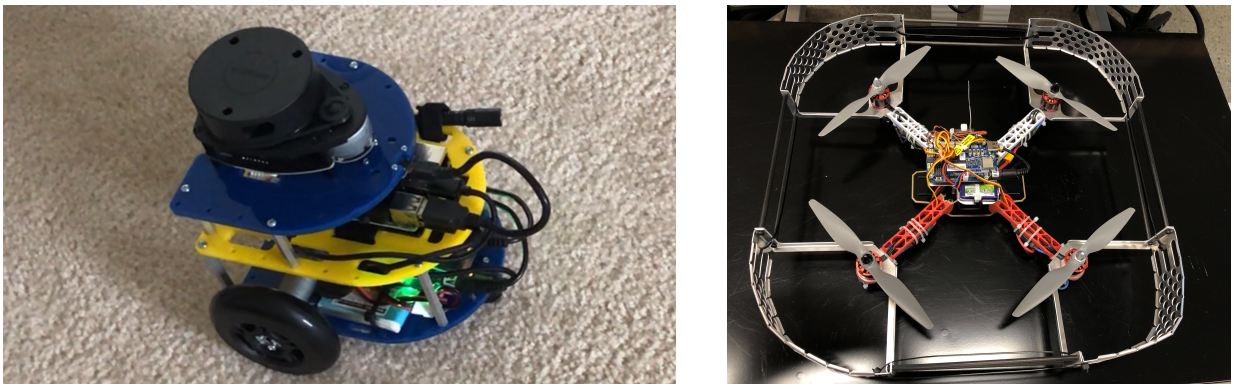

Figure 3: The M-Bot Mini Ground Robot and the Quadcopter, used in my laboratory for running MARVeLus on hardware

The M-Bot Mini is a small differential-drive wheeled robot, typically used as a teaching platform for robotics classes, including ROB 550 Robotic Systems Laboratory at the University of Michigan. On-board sensors include a 9-axis inertial measurement unit (IMU), rotating LiDAR, and a front-facing optical camera. The drivetrain consists of two geared DC motors with built-in magnetic quadrature encoders for measuring their speed and position. The robot is equipped with a Beaglebone Blue, a Linux-capable single-board computer featuring built-in motor controllers and quadrature encoder inputs. If needed, the Beaglebone Blue can work in tandem with a Raspberry Pi which handles processing of LiDAR and camera data.

In our configuration, the Beaglebone Blue serves as the main processor that communicates with the MARVeLus runtime and runs low-level controller code. The controller code interprets read or write operations from the MARVeLus runtime as sensor or actuator actions and is designed to be as simple as possible to minimize

unverified code in the stack. In our current implementation, the controller code includes closed-loop motor speed control allowing the robot to simulate the maneuvering characteristics of various other vehicles ?at-scale?.

Research Question: How to enable verification, simulation and execution on a small ground robot in the laboratory?

The MARVeLus runtime can be configured to run remotely on a separate computer or locally on-board the robot. In the former configuration, the computer running the MARVeLus runtime and user code communicates wirelessly with the robot over the WLAN network and delegates sensor and actuator actions to the robot when needed. This configuration enables easier debugging, including substituting the robot with a simulator for offline development. In the latter configuration, the MARVeLus runtime and user code execute directly on the robot which allows for reduced communications latency, though a connection is still established with another computer for monitoring and debugging. In both configurations, the MARVeLus runtime communicates with the controller code via the Lightweight Communications and Marshalling (LCM) protocol through a series of ?store? and ?read? commands which aim to replicate the robot-relevant state variables in the MARVeLus runtime with those present on the actual (or simulated) robot.

Task 3.2: Quadcopter in laboratory

Research Question: How to transfer the knowledge gained on a ground robot to an autonomous quadcopter?

The M-330 Quadrotor is an aerial robot targeted by MARVeLus for initial testing and execution. M-330 was derived from a hobbyist quadrotor platform and is used by several other groups at our institution, allowing for future collaboration opportunities and continual improvements to the design. It features modular components and ample payload capacity, making it a flexible platform adaptable to a variety of experiments. Additionally, the inclusion of propeller guards enhances its safety when operating indoors.

As with the M-Bot Mini, a Beaglebone Blue serves as the main compute module onboard the M-330, where its wireless communications, real-time capabilities, and sensors allow for a highly integrated, easily configurable flight controller. Having a common compute platform between robots in our project means that much of the core MARVeLus functionality can be shared, allowing us to concentrate our development efforts on platform-specific features.

Along with the sensors on the Beaglebone Blue, the quadrotor can also be equipped with reflective targets to make use of motion capture in both indoor and outdoor flight test facilities. This provides us with more accurate attitude and position data than can be collected using onboard sensors alone. Since motion capture may not be available in situations where quadrotors are used, we intend to eventually move towards a fully-onboard sensor suite. Nevertheless, the use of motion capture in initial testing provides a reliable point of reference that is easier to model.

Preliminary Work. Our recent paper on quadcopter collision avoidance [3] provides safety regions in that context, formally proved in the KeYmaera X theorem prover. Implementing those safe regions with real hardware will show us how applicable this proof is with actual hardware components and physics.

Task 3.3: Aircraft Collision Avoidance and ACAS X

The initial motivation for this research program stems from unanswered questions from prior work on formally verifying the aircraft collision avoidance system ACAS X. Because I have done extensive work on the initial verification of ACAS X, I have a deep understanding of the intricacies of the system and of its verification. We will adapt the reference implementation of ACAS X to use MARVeLus, as well as equip it with the differential equations modeling simplified flight dynamics of the aircraft. Finally, we will perform an end-to-end verification of ACAS X in MARVeLus, and we may discover issues along the way.

Research Question: How to provide a verified and executable *reference implementation* for the ACAS X collision avoidance system?

Throughout this case study, I will leverage past interactions with different actors of the ACAS X project, including with the Federal Aviation Administration program director and the main designers of the system.

Task 3.4: Emergency collision avoidance maneuvers in cars

I have recently had funding from the Toyota Research Institute (TRI) for formally verifying the safety of evasive maneuvers that use braking, turning or a combination of braking and turning. In that work [2, 1], there exist similar gaps between the model that is verified and the code that really executes on the car. My liaison at TRI has shown interest in verifying properties from end-to-end, with potential integration into an internal research product. This collaboration will ensure industrial feedback, technology transfer, as well as the opportunity for follow-up funding beyond this CAREER grant.

Task 3.5: Incorporation in AERO 552 Aerospace Information Systems

In my graduate class AERO 552 on Aerospace Information Systems, one module is dedicated to formal verification. The current associated homework is purely theoretical, and as a result students only get a partial view on the applicability of formal verification in their future jobs. Once a stable prototype is built, I would like to incorporate the use of MARVeLus in one or two homeworks of AERO 552, so that students can understand how to use verification tools in practice. The case study we will ask them to implement will be a simplified version of our ACAS X work, where we envision to use very simplified dynamics, e.g., using infinite acceleration and instantaneous change of direction.

As an added advantage, the incorporation in AERO 552 will give us a first set of users, which will help us identify shortcomings in terms of usability and performance. This will be very useful for the development of MARVeLus, and will also help with our Evaluation plan.

7 Evaluation Plan

Our evaluation plan is three-fold: experiments on small robots in the laboratory (tasks 3.1 and 3.2), programming of industrical applications (tasks 3.3 and 3.4), and finally use of the MARVeLus language in class with graduate students (task 3.5).

Industry adoption. Finally, an essential part of our evaluation will be the technology transfer to industry. Through our collaborations with the different industries Notably, we will work hard so that our reference implementation

8 Risks and Mitigations

We place ourselves at a lower level of abstraction than

9 Broader Impacts

9.1 Education and Outreach

Educating students to design and program correct cyber-physical systems is important at all levels, from middle school to graduate school. Our education plan targets three populations: middle school students for an initiation to safety and verification, late-stage undergraduate students and early-stage graduate students through the organization of mentoring workshops, and early-stage to established graduate students through the organization of the Midwest Programming Languages Seminar.

9.1.1 Collaboration with local middle school

[JB: – IMPORTANT – This might fall through. Bill Van Loo has left STEAM, and his replacement Sarah Van Loo has been unresponsive to my emails. Still trying to get her to answer, or might consider another school. Currently in talks with the school district.] I will collaborate with Sarah Van Loo, technology and engineering teacher at the STEAM Northside middle school in Ann Arbor, Michigan (see collaboration letter). The STEAM middle school, standing for "Science, Technology, Engineering, Arts and Mathematics", is a unique integration of traditional STEM fields with the Arts. It provides an environment where students, staff, and the community are actively engaged in project-based learning through a student-centered, integrated approach with real-world applications. It attracts students from every part of the school district of Ann Arbor.

Sarah Van Loo is currently teaching at the middle school following the Project Lead The Way (`www.pltw.org`) from 6th to 8th grade. We will collaborate in the "Automation and Robotics" class to enhance the student's aware-

ness with safety of critical systems such as self-driving cars, and familiarize the students to the concept of verification.

During the collaboration, I will first give a presentation regarding safety and verification, targeted to middle schoolers, followed by some hands-one experience on an autonomous drag racer project. The presentation will show the students how to use simple geometric arguments from their mathematics class to develop conditions ensuring collision safety of their drag racers, from data provided by camera and proximity sensors. Together with the teachers, I will then show them how to implement those conditions, and the students will be able to test them in their lab using new camera sensors purchased with the NSF grant. The hands-on experience will be done in collaboration with my graduate students, giving the middle school students a chance to discuss, interact, and establish longer-term connections with graduate students. The end goal of the project is to give students a greater awareness of the importance of safety in designing autonomous systems, and the realization that scientific tools exist to ensure this safety.

The purchase of new camera sensors will be essential to the successful completion of this project, as it will allow students to detect and identify foreign objects, then implement safety conditions and see them in action. Essentially, it will allow the drag racer projects to behave like small autonomous cars that the students need to keep safe from collisions with other objects.

In the long run, the middle school teacher and I will establish new orientations and objectives of the class. They will be able to give back to Project Lead The Way so that safety and verification can be introduced to middle school students across many more middle and high schools across the country. A focus on safety early on is becoming increasingly important as some of those students will grow to become the engineers and designers of the delivery drones and self-driving cars of tomorrow.

9.1.2 Organization of Mentoring Workshops

I have a history of organizing mentoring workshops, especially targeted to underrepresented minorities. I was an organizer of the 2020 and 2021 Verification Mentoring Workshops (VMW). VMW is a workshop for students held the first two days of the Computer-Aided Verification (CAV) conference, in July. I was a member of the team in 2020, and a co-chair of the workshop in 2021. The purpose of this yearly workshop is to provide mentoring and career advice to early-stage graduate students and late-stage undergraduate students, introduce them to research topics aligned with the CAV conference, give them practical advice on graduate school, and provide them with resources and contacts to be successful in their graduate career. The workshop particularly encourages participation of women and underrepresented minorities. Due to the Covid-19 pandemic, both the 2020 and 2021 workshop was held online, but still attracted more than 400 registrants, and 50 to 60 participants at each session. Of those 435 registrants in 2020, 95 (22%) identified as female and 3 (0.7%) identified as nonbinary; 6 registrants (1.4%) identified as Black, African or African-American, and 3 (0.7%) identified as Hispanic. As part of a 2020 workshop, the other organizers and I introduced a survey of the CAV community, answered by over 200 researchers, to better understand the community, its shortcomings and struggles. The results of the survey were shared with the VMW audience. In the long run, I am planning to stay involved in the organization of mentoring workshops, including both VMW and the Programming Languages Mentoring Workshops (PLMW).

9.1.3 Organization of the Midwest Programming Languages Seminar (MWPLS)

I am also organizing the next Midwest Programming Languages Workshop (MWPLS). The workshop was initially scheduled to be held in the Fall of 2020, but due to the Covid-19 pandemic has been delayed to either the Winter of 2021 or the Fall of 2021. This regional wokshop offers a friendly venue for graduate students of all seniorities and backgrounds to present in-progress work to their peers from institutions in the Midwestern United States. We particularly encourage presentations and posters from women and students from underrepresented minorities, and students who are still early in their PhD studies. For many young graduate students, their MWPLS presentation is their first talk outside of their home university, and an essential part of their career development. For faculty and researchers, it is a small regional venue encouraging local Programming Languages connections in the Midwest.

9.2 Industrial Collaborations

I have established collaborations with the Toyota Research Institute, Collins Aerospace, the National Aeronautics and Space Administration (NASA), and Amazon Web Services, through past or present funding provided by those entities. I have visited Toyota in Palo Alto, CA multiple times, Collins in Minneapolis, MN in February 2019, and I am spending a week at NASA Langley Research Center in July 2022. I am also currently in discussions with Boeing to establish a similar collaboration. I will use those collaborations as a source of inspiration for relevant problems, and as a source of challenges and relevant industrial applications. I will continue to seek relevant industrial collaborations through his participation to workshops and short research visits.

9.3 Dissemination via Publications and Open-Source software

Due to the multidisciplinary nature of this research program, its results will be published in the leading conferences and journals of four different communities: the programming languages community (POPL, PLDI, ICFP, SPLASH), the hybrid and cyber-physical systems community (HSCC, ICCPS, ADHS), the logic and verification community (CAV, TACAS, IJCAR, CADE, LICS), and the embedded software community (EMSOFT). I will use my website to disseminate research material and software, and the pre-prints of the papers will be available online. Starting the third year of the proposal, I plan on organizing workshops or tutorials at CPSWeek (grouping HSCC and ICCPS) and at the SPLASH conference to bring together members of the programming languages and cyber-physical systems community.

The software associated with this research program will be released under the liberal modified BSD license, making it free to use for any purpose, including unrestricted academic and commercial use, incorporation in other tools, and modifications. It will be disseminated through hosting services such as GitHub or BitBucket. This enables quick translation of the results to research and engineering communities, as well as an opportunity for other industrial and academic researchers to provide feedback and suggestions for improvements. I will advocate open-source development among the undergraduate and graduate students working with him.

9.4 Graduate Education and Research

I teach the course AERO 552: Aerospace Information Systems. I already offered this graduate course four times in Fall 2017, Winter 2019, Winter 2021 and Winter 2022. I have introduced aspects of formal verification in the curriculum of AERO 552, including aspects of certification of aerospace software. Once a prototype compiler for MARVeLus is available, I will introduce MARVeLus in class and in lab to AERO 552 students, thereby giving them exposure to the latest research in the field, but also giving the MARVeLus development team valuable feedback.

The majority of the funds requested for this project will be used to support graduate students in the form of graduate research assistantships. I will primarily recruit students from the Aerospace Engineering, Computer Science and Engineering, and Robotics Programs. I currently advise or co-advise four graduate students: one in Aerospace Engineering, one in Computer Science and Engineering, and two in Robotics.

9.5 Undergraduate Education and Research

The research outcome of this effort will also be integrated into the courses that I will offer in his department in coming years. Last Fall I have created a completely new class AERO 350: Fundamentals of Aerospace Computing. This junior-level class gives students a firm understanding of the use of computing in aerospace engineering; after spending 6 weeks on the fundamentals of programming, it branches out into 4 weeks on computational science followed by 4 weeks on embedded systems. The class was limited to a small enrollment of 7 for its first offering, but was a great success with the students, and I received very encouraging instructor-evaluation score of 4.8 out of 5.0 possible, above average. Enrollment for Fall 2019 is already at 24. The class is central to the evolution of the teaching philosophy of the department to embrace the computer revolution of aerospace engineering, and it has recently been voted by the faculty as mandatory for all undergraduates in aerospace engineering starting in the Fall of 2020. I am planning to continue to offer AERO 350, and update and modify AERO 552 so the two courses can form a sequence.

Once MARVeLus will be well-established in the graduate class AERO 552, we will launch a simpler lab in one of the last lectures of AERO 350, again giving our students access to the latest research. As a byproduct of this

introduction, we will certainly get valuable feedback from less experienced programmers than in the graduate class.

Recruiting and mentoring of underrepresented groups I am dedicated to broadening participation in engineering education and creating a more diverse engineering community. Every year I recruit undergraduate student researchers through the Summer Undergraduate Research in Engineering (SURE) program, and three of the mentored undergraduates have already written four conference papers in less than two years [2, 1, 3, 64]. Additionally, among the graduate students he has advised, one is a woman and one is a refugee from Iran. Finally, the cross-disciplinary nature of this research program will attract students with a diverse set of backgrounds and enrolled in different programs, and their mutual interactions will further enrich their graduate experience.

10 Implementation Plan and Timeline

The timeline of the proposed work including some of the milestones is given below.

Year 1: Initial design of the language MARVeLus and its semantics (Tasks 1.1 and 1.2). Initial experiments with case studies on ACAS X and car collision avoidance to see what the program and specifications would look like. Recruitment of a graduate student and an undergraduate researcher to work on the project. Along with teacher Sarah Van Loo, design precise participation in the STEAM middle school; first participation in the middle school class at STEAM middle school.

Year 2: Finalize the design of MARVeLus and of its semantics (Tasks 1.1 and 1.2). Initial interpreter for the executable part of MARVeLus (Task 3.1). Design the refinement type system of MARVeLus enabling formal verification, and implement the verification condition generation (Task 2.1). Extend the refinement type system to enable verification of safety and liveness properties (Task 2.2). Second participation in the middle school class at STEAM middle school

Year 3: Compiler (first implementation, unverified) for the executable part of MARVeLus (Task 3.1). Development of the simulation engine of MARVeLus (Task 2.3). Initial attempts with case studies of ACAS X and car collision avoidance, proofs for simple cases (Tasks 4.1 and 4.2). Organize workshop at CPSWeek or SPLASH. Third participation in the middle school class at STEAM middle school; return on experience after three editions and giving back material to Project Lead The Way.

Year 4: Start formally verifying the compiler (Task 3.2). Case study of end-to-end formal verification of ACAS X (Task 4.1). Create a lecture and a lab on MARVeLus in AERO 552 (graduate level) and have the students use it in class. Fourth participation in the middle school class at STEAM middle school.

Year 5: Finish formally verifying the compiler (Task 3.2). Complete case study of emergency collision avoidance maneuvers for cars (Task 4.2). Fifth participation in the middle school class at STEAM middle school; ensuring that the material will continue being covered beyond the grant, and that all the lessons learnt have been transferred back to the Project Lead The Way program to use by other middle schools. Continue teaching MARVeLus in AERO 552 (graduate level), and create an undergraduate version to be taught at the end of AERO 350 (undergraduate level).

11 Results from Previous NSF Support

Jean-Baptiste Jeannin: #2219997: FMitF: Track I: Foundational Approaches for End-to-end Formal Verification of Computational Physics, $750,000, 10/01/22-09/30/26. PI: Jean-Baptiste Jeannin, co-PI: Karthik Duraisamy. **Intellectual Merit.** Numerical solutions of differential equations, accompanied by techniques to quantify uncertainties are widely used in analysis, design and other decision-making tasks in science and engineering. Examples include modeling climate change, discovering new materials, designing aircraft, and understanding the early universe. This simulation process, however, involves errors arising from a number of sources such as the finiteness of the numerical discretization, potential lack of convergence of algorithms, floating-point arithmetic, and sampling required to quantify uncertainties. The goal of this work is to develop theory and formalisms to ensure a rigorous handle over errors and uncertainties. The current state of the art in numerical analysis relies on paper proofs; in contrast our proofs are mechanically checked in an interactive theorem prover, and provide end-to-end guarantees from the problem expressed on a sheet of paper to the implementation at the C code level, and down to the executable code that computes numerical results. **Broader Impacts.** The proposed activities – centered around

formal verification – have the potential to re-invigorate existing links between theoretical computer science and computational science, and bridge some of the gaps. Our work will open a new space of physical applications to formal methods researchers, and for computational physicists to see the value in formalizing their programs. We will organize workshops to promote formal verification of computational physics, to bring together formal methods practitioners and computational physicists. PI Jeannin has created a course that is now mandatory for Aerospace Engineering undergraduates that is unique in its integration of computer science fundamentals as well as computational physics methods, aligned with the spirit of the present grant.

[JB: Some general remarks / questions:

- Are there too many applications (currently five)? Currently I have two in the lab (M-bot ground robot, quad-copter), two industrial (airplanes ACAS X, cars, both extensions of previous work), and one in class. If I had to pick 3, I would probably just do the ground robot, ACAS X and the class. Should I just focus on those 3?
- I'm still not so happy with the title, it's too long and bulky. Maybe I should focus more on verification and execution, since the simulation part has already been done by Zelus. Ideas:
  - old title: CAREER: A Programming Language for the Verification, Execution and Simulation of Cyber-Physical Systems
  - MARVeLus: Unifying Verification and Execution of Cyber-Physical Systems [drops simulation]
  - MARVeLus: a Verifiable and Executable Programming Language for CPS [drops simulation]
  - Verifiable Execution and Simulation of Cyber-Physical Systems
  - Keywords: Unification, Cyber-Physical Systems, Hybrid Systems, Verification, Execution, Simulation
- I have dropped the part about verified compilation; I only mention it in passing. I would be nice to have, but after several feedbacks regarding it, I think it's too much for this grant which is essentially the same budget as an NSF Small. Plus the Zélus team has done most of it already.
- Related to the previous
- On the ACAS X example, the core of the verification involves a lookup table. I don't talk explicitly about how to deal with lookup tables, but I make ACAS X a front-and-center case study.
- Is providing a verified reference implementation for ACAS X too ambitious? At the same time we have 5 years, and it's a really nice, applicable case study.
- do we keep simulations? After all most of the work is verif+exec, whereas simulation piggy-backs on the work done already by Zelus. The only work that is really doing simulation is the relationship with Coq proofs of numerical methods for ODEs.
- In the past Wes has suggested human studies. I am not sure if it's a good idea, especially because I've never done it, hence low credibility in actually implementing them, and I would have to get some pointers on how to do it specifically.

]

[JB: High-level feedback from NSF reviewers that I have tried to address:

- Not enough details – needs much more meat
- No Evaluation Plan
- Add concrete examples

]

References

[1] A. Abhishek, H. Sood, and J.-B. Jeannin. Formal verification of braking while swerving in automobiles. In *Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control*, pages 1–11, 2020.

[2] A. Abhishek, H. Sood, and J.-B. Jeannin. Formal verification of swerving maneuvers for car collision avoidance. In *2020 American Control Conference (ACC)*. IEEE, 2020.

[3] E. Adler and J.-B. Jeannin. Formal verification of collision avoidance for turningmaneuvers in uavs. In *AIAA Aviation 2019 Forum*. AIAA, 06 2019.

[4] M. Althoff, O. Stursberg, and M. Buss. Computing reachable sets of hybrid systems using a combination of zonotopes and polytopes. *Nonlinear analysis: hybrid systems*, 4(2):233–249, 2010.

[5] R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid systems*, pages 209–229. Springer, 1992.

[6] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. Netkat: Semantic foundations for networks. In *Principles of Programming Languages (POPL)*, volume 49, pages 113–126. ACM, 2014.

[7] Y. Annpureddy, C. Liu, G. Fainekos, and S. Sankaranarayanan. S-taliro: A tool for temporal logic falsification for hybrid systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 254–257. Springer, 2011.

[8] A. W. Appel. Verified software toolchain. In G. Barthe, editor, *Programming Languages and Systems*, pages 1–17, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[9] A. W. Appel and Y. Bertot. C floating-point proofs layered with vst and flocq. *Journal of Formalized Reasoning*, 13(1):1?16, Jan. 2020.

[10] S. N. Artëmov, J. M. Davoren, and A. Nerode. Topological semantics for hybrid systems. In *Logical Foundations of Computer Science, 4th International Symposium, LFCS'97, Yaroslavl, Russia, July 6-12, 1997, Proceedings*, pages 1–8, 1997.

[11] L. Augustsson. Cayenne—a language with dependent types. In *ACM SIGPLAN Notices*, volume 34, pages 239–250. ACM, 1998.

[12] K. Bae, P. C. Ölveczky, T. H. Feng, and S. Tripakis. Verifying ptolemy II discrete-event models using real-time maude. In *International Conference on Formal Engineering Methods*, pages 717–736. Springer, 2009.

[13] S. Bak, S. Bogomolov, and T. T. Johnson. Hyst: a source transformation and translation tool for hybrid automaton models. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, pages 128–133. ACM, 2015.

[14] T. Ball and S. K. Rajamani. The slam project: debugging system software via static analysis. In *ACM SIGPLAN Notices*, volume 37, pages 1–3. ACM, 2002.

[15] G. B. Banusić, R. Majumdar, M. Pirron, A.-K. Schmuck, and D. Zufferey. Pgcd: robot programming and verification with geometry, concurrency, and dynamics. In *Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems*, pages 57–66. ACM, 2019.

[16] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliatre, E. Gimenez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, and B. Werner. The Coq proof assistant reference manual: Version 6.1. 1997.

[17] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *International Conference on Computer Aided Verification*, pages 171–177. Springer, 2011.

[18] G. Berry. Scade: Synchronous design and validation of embedded control software. In *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, pages 19–33. Springer, 2007.

[19] B. Bohrer, Y. K. Tan, S. Mitsch, M. O. Myreen, and A. Platzer. Veriphy: Verified controller executables from verified cyber-physical system models. In *Programming Language Design and Implementation (PLDI), ACM SIGPLAN Notices*, volume 53, pages 617–630. ACM, 2018.

[20] T. Bourke, L. Brun, P.-É. Dagand, X. Leroy, M. Pouzet, and L. Rieg. A formally verified compiler for lustre. In *Programming Language Design and Implementation (PLDI), ACM SIGPLAN Notices*, volume 52, pages 586–601. ACM, 2017.

[21] T. Bourke and M. Pouzet. Zélus: A synchronous language with ODEs. In *16th International Conference on Hybrid Systems: Computation and Control (HSCC'13)*, pages 113–118, Philadelphia, USA, Mar. 2013.

[22] F. Boussinot and R. De Simone. The esterel language. *Proceedings of the IEEE*, 79(9):1293–1304, 1991.

[23] A. Bove, P. Dybjer, and U. Norell. A brief overview of agda–a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics*, pages 73–78. Springer, 2009.

[24] C. Casinghino, V. Sjöberg, and S. Weirich. Combining proofs and programs in a dependently typed language. In *ACM SIGPLAN Notices*, volume 49, pages 33–45. ACM, 2014.

[25] X. Chen, E. Abraham, and S. Sankaranarayanan. Taylor model flowpipe construction for non-linear hybrid systems. In *Real Time Systems Symposium (RTSS)*, pages 183–192. IEEE Press, 2012.

[26] X. Chen, E. Abraham, and S. Sankaranarayanan. Flow*: An analyzer for non-linear hybrid systems. In *Computer-Aided Verification (CAV)*, volume 8044 of *Lecture Notes in Computer Science*, pages 258–263. Springer-Verlag, 2013.

[27] C.-H. Cheng, T. Fristoe, and E. A. Lee. Applied verification: The Ptolemy approach. 2008.

[28] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith. *Model checking*. MIT press, 2018.

[29] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C. In *International Conference on Software Engineering and Formal Methods*, pages 233–247. Springer, 2012.

[30] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer.

[31] P. Derler, E. A. Lee, M. Torngren, and S. Tripakis. Cyber-physical system design contracts. In *ICCPS '13: ACM/IEEE 4th International Conference on Cyber-Physical Systems*, April 2013.

[32] A. Donzé. Breach, a toolbox for verification and parameter synthesis of hybrid systems. In *International Conference on Computer Aided Verification*, pages 167–170. Springer, 2010.

[33] F.-X. Dormoy. Scade 6: a model based solution for safety critical software development. In *Proceedings of the 4th European Congress on Embedded Real Time Software (ERTS?08)*, pages 1–9, 2008.

[34] G. Dowek, C. Muñoz, and V. Carreño. Provably safe coordinated strategy for distributed conflict resolution. In *AIAA Guidance Navigation, and Control Conference and Exhibit*, 2005.

[35] J.-C. Filliâtre and C. Marché. The why/krakatoa/caduceus platform for deductive program verification. In *International Conference on Computer Aided Verification*, pages 173–177. Springer, 2007.

[36] M. J. Fischer and R. E. Ladner. Propositional dynamic logic of regular programs. *Journal of computer and system sciences*, 18(2):194–211, 1979.

[37] R. W. Floyd. Assigning meanings to programs. In *Program Verification*, pages 65–81. Springer, 1967.

[38] G. Frehse, C. L. Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. Spaceex: Scalable verification of hybrid systems. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 379–395, 2011.

[39] D. J. Fremont, T. Dreossi, S. Ghosh, X. Yue, A. L. Sangiovanni-Vincentelli, and S. A. Seshia. Scenic: a language for scenario specification and scene generation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 63–78. ACM, 2019.

[40] P. Fritzson. *Principles of object-oriented modeling and simulation with Modelica 2.1*. John Wiley & Sons, 2010.

[41] N. Fulton, S. Mitsch, J.-D. Quesel, M. Völp, and A. Platzer. KeYmaera X: An axiomatic tactical theorem prover for hybrid systems. In A. Felty and A. Middeldorp, editors, *CADE*, volume 9195 of *LNCS*, pages 527–538. Springer, 2015.

[42] A. Galdino, C. Muñoz, and M. Ayala. Formal verification of an optimal air traffic conflict resolution and recovery algorithm. In *WoLLIC*, volume 4576 of *LNCS*. Springer, 2007.

[43] K. Ghorbal, J.-B. Jeannin, E. Zawadzki, A. Platzer, G. J. Gordon, and P. Capell. Hybrid theorem proving of aerospace systems: Applications and challenges. *Journal of Aerospace Information Systems (JAIS)*, 11(10):702–713, 2014.

[44] K. Ghorbal and A. Platzer. Characterizing algebraic invariants by differential radical invariants. In E. Ábrahám and K. Havelund, editors, *TACAS*, volume 8413, pages 279–294. Springer, 2014.

[45] K. Ghorbal, A. Sogokon, and A. Platzer. A hierarchy of proof rules for checking positive invariance of algebraic and semi-algebraic sets. *Computer Languages, Systems and Structures*, 47(1):19–43, 2017.

[46] A. Guha, J.-B. Jeannin, R. Nigam, J. Tangen, and R. Shambaugh. Fission: Secure dynamic code-splitting for javascript. In *2nd Summit on Advances in Programming Languages (SNAPL)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[47] S. Gulwani and A. Tiwari. Constraint-based approach for analysis of hybrid systems. In *International Conference on Computer Aided Verification*, pages 190–203. Springer, 2008.

[48] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[49] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, Cambridge, MA, 2000.

[50] J. Harrison. Hol light: An overview. In *International Conference on Theorem Proving in Higher Order Logics*, pages 60–66. Springer, 2009.

[51] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. Hytech: A model checker for hybrid systems. In *International Conference on Computer Aided Verification*, pages 460–463. Springer, 1997.

[52] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast. In *International SPIN Workshop on Model Checking of Software*, pages 235–239. Springer, 2003.

[53] H. Herencia-Zapana, J.-B. Jeannin, and C. Muñoz. Formal verification of safety buffers for state-based conflict detection and resolution. In *27th International Congress of the Aeronautical Sciences (ICAS)*, 2010.

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

[55] J.-B. Jeannin, G. de Caso, J. Chen, Y. Gurevich, P. Naldurg, and N. Swamy. DKAL*: Constructing executable specifications of authorization protocols. In *International Symposium on Engineering Secure Software and Systems (ESSoS)*, pages 139–154. Springer, 2013.

[56] J.-B. Jeannin, K. Ghorbal, Y. Kouskoulas, R. Gardner, A. Schmidt, E. Zawadzki, and A. Platzer. Formal verification of acas x, an industrial airborne collision avoidance system. In *Proceedings of the 12th International Conference on Embedded Software (EMSOFT)*, pages 127–136. IEEE Press, 2015.

[57] J.-B. Jeannin, K. Ghorbal, Y. Kouskoulas, R. Gardner, A. Schmidt, E. Zawadzki, and A. Platzer. A formally verified hybrid system for the next-generation airborne collision avoidance system. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 21–36. Springer, 2015.

[58] J.-B. Jeannin, K. Ghorbal, Y. Kouskoulas, A. Schmidt, R. Gardner, S. Mitsch, and A. Platzer. A formally verified hybrid system for safe advisories in the next-generation airborne collision avoidance system. *International Journal on Software Tools for Technology Transfer (STTT)*, 19(6):717–741, 2017.

[59] J.-B. Jeannin, D. Kozen, and A. Silva. Language constructs for non-well-founded computation. In *European Symposium on Programming (ESOP)*, pages 61–80. Springer, 2013.

[60] J.-B. Jeannin, D. Kozen, and A. Silva. Cocaml: Functional programming with regular coinductive types. *Fundamenta Informaticae*, 150(3-4):347–377, 2017.

[61] J.-B. Jeannin, D. Kozen, and A. Silva. Well-founded coalgebras, revisited. *Mathematical Structures in Computer Science (MSCS)*, 27(7):1111–1131, 2017.

[62] J.-B. Jeannin and A. Platzer. dtl^2: Differential temporal dynamic logic with nested temporalities for hybrid systems. In *International Joint Conference on Automated Reasoning (IJCAR)*, pages 292–306. Springer, 2014.

[63] R. Jobredeaux, T. E. Wang, and E. M. Feron. Autocoding control software with proofs i: Annotation translation. In *2011 IEEE/AIAA 30th Digital Avionics Systems Conference*, pages 7C1–1. IEEE, 2011.

[64] K. D. Julian, S. Sharma, J.-B. Jeannin, and M. J. Kochenderfer. Verifying aircraft collision avoidance neural networks through linear approximations of safe regions. In *Verification of Neural Networks (VNN)*, 2019.

[65] N. Kheterpal, E. Tang, and J.-B. Jeannin. Automating geometric proofs of collision avoidance with active corners. In *Formal Methods in Computer-Aided Design*, 2022, to appear.

[66] M. J. Kochenderfer and J. P. Chryssanthacopoulos. Robust airborne collision avoidance through dynamic programming. Technical Report ATC-371, MIT Lincoln Laboratory, January 2010.

[67] M. J. Kochenderfer and N. Monath. Compression of optimal value functions for Markov decision processes. In *Data Compression Conference*, Snowbird, Utah, 2013.

[68] Y. Kouskoulas, D. Genin, A. Schmidt, and J.-B. Jeannin. Formally verified safe vertical maneuvers for non-deterministic, accelerating aircraft dynamics. In *International Conference on Interactive Theorem Proving (ITP)*, pages 336–353. Springer, 2017.

[69] D. Kozen. Results on the propositional μ-calculus. *Theoretical computer science*, 27(3):333–354, 1983.

[70] D. Kozen. Kleene algebra with tests. *Transactions on Programming Languages and Systems*, 19(3):427–443, May 1997.

[71] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, 1994.

[72] R. Lee, M. J. Kochenderfer, O. J. Mengshoel, G. P. Brat, and M. P. Owen. Adaptive stress testing of airborne collision avoidance systems. In *Digital Avionics Systems Conference (DASC), 2015 IEEE/AIAA 34th*, pages 6C2–1. IEEE, 2015.

[73] P. LeGuernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with signal. *Proceedings of the IEEE*, 79(9):1321–1336, 1991.

[74] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370. Springer, 2010.

[75] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. The ocaml system release 4.02. *Institut National de Recherche en Informatique et en Automatique*, 54, 2014.

[76] S. M. Loos, D. W. Renshaw, and A. Platzer. Formal verification of distributed aircraft controllers. In *HSCC*, pages 125–130. ACM, 2013.

[77] R. Majumdar, A. Mathur, M. Pirron, L. Stegner, and D. Zufferey. Paracosm: A language and tool for testing autonomous driving systems. *arXiv preprint arXiv:1902.01084*, 2019.

[78] O. Maler and D. Nickovic. Monitoring temporal properties of continuous signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pages 152–166. Springer, 2004.

[79] Z. Manna and A. Pnueli. *Temporal verification of reactive systems: safety*. Springer Science & Business Media, 2012.

[80] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The definition of standard ML: revised*. MIT press, 1997.

[81] C. Muñoz, A. Narkawicz, and J. Chamberlain. A TCAS-II resolution advisory detection algorithm. In *Proceedings of the AIAA Guidance Navigation, and Control Conference and Exhibit 2013*, number AIAA-2013-4622, Boston, Massachusetts, August 2013.

[82] A. Narkawicz and C. Muñoz. Formal verification of conflict detection algorithms for arbitrary trajectories. *Reliable Computing*, 17:209–237, December 2012.

[83] A. Narkawicz and C. Muñoz. A formally verified conflict detection algorithm for polynomial trajectories. In *Proceedings of the 2015 AIAA Infotech @ Aerospace Conference*, Kissimmee, Florida, January 2015.

[84] A. J. Narkawicz, C. A. Muñoz, J. M. Upchurch, J. P. Chamberlain, and M. C. Consiglio. A well-clear volume based on time to entry point. Technical Memorandum NASA/TM-2014-218155, NASA, Langley Research Center, Hampton VA 23681-2199, USA, January 2014.

[85] J. L. Newcomb, S. Chandra, J.-B. Jeannin, C. Schlesinger, and M. Sridharan. Iota: A calculus for internet of things automation. In *ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, pages 119–133. ACM, 2017.

[86] P. Nilsson, O. Hussien, Y. Chen, A. Balkan, M. Rungger, A. Ames, J. Grizzle, N. Ozay, H. Peng, and P. Tabuada. Preliminary results on correct-by-construction control software synthesis for adaptive cruise control. In *53rd IEEE Conference on Decision and Control*, pages 816–823. IEEE, 2014.

[87] P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of real-time maude. *Higher-order and symbolic computation*, 20(1-2):161–196, 2007.

[88] C.-M. Ong. *Dynamic simulation of electric machinery: using MATLAB/SIMULINK*, volume 5. Prentice Hall PTR Upper Saddle River, NJ, 1998.

[89] S. Owre, J. M. Rushby, and N. Shankar. Pvs: A prototype verification system. In *International Conference on Automated Deduction*, pages 748–752. Springer, 1992.

[90] L. C. Paulson. *Isabelle: A generic theorem prover*, volume 828. Springer Science & Business Media, 1994.

[91] A. Platzer. Differential dynamic logic for hybrid systems. *J. Autom. Reas.*, 41(2):143–189, 2008.

[92] A. Platzer. The complete proof theory of hybrid systems. In *2012 27th Annual IEEE Symposium on Logic in Computer Science*, pages 541–550. IEEE, 2012.

[93] A. Platzer and E. M. Clarke. Formal verification of curved flight collision avoidance maneuvers: A case study. In *FM*, volume 5850 of *LNCS*, pages 547–562. Springer, 2009.

[94] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (FOCS 1977)*, pages 46–57. IEEE, 1977.

[95] A. Pnueli and Z. Manna. The temporal logic of reactive and concurrent systems: specification. *Springer*, 16:12, 1992.

[96] C. Ptolemaeus. *System design, modeling, and simulation: using Ptolemy II*, volume 1. Ptolemy. org Berkeley, 2014.

[97] V. Raman, A. Donzé, M. Maasoumy, R. M. Murray, A. Sangiovanni-Vincentelli, and S. A. Seshia. Model predictive control with signal temporal logic specifications. In *53rd IEEE Conference on Decision and Control*, pages 81–87. IEEE, 2014.

[98] V. Raman, A. Donzé, D. Sadigh, R. M. Murray, and S. A. Seshia. Reactive synthesis from signal temporal logic specifications. In *Proceedings of the 18th international conference on hybrid systems: Computation and control*, pages 239–248. ACM, 2015.

[99] T. Ramananandro, P. Mountcastle, B. Meister, and R. Lethin. A unified coq framework for verifying c programs with floating-point computations. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2016, page 15?26, New York, NY, USA, 2016. Association for Computing Machinery.

[100] H. Ravanbakhsh and S. Sankaranarayanan. Counter-example guided synthesis of control lyapunov functions for switched systems. In *IEEE Control and Decision Conference (CDC)*, pages 4232–4239. IEEE Press, 2015.

[101] L. Ryzhyk, N. Bjørner, M. Canini, J.-B. Jeannin, C. Schlesinger, D. B. Terry, and G. Varghese. Correct by construction networks using stepwise refinement. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 683–698, 2017.

[102] S. Sankaranarayanan, X. Chen, and E. Abraham. Lyapunov function synthesis using handelman representations. In *IFAC conference on Nonlinear Control Systems (NOLCOS)*, pages 576–581, 2013.

[103] S. Sankaranarayanan and A. Tiwari. Relational abstractions for continuous and hybrid systems. In *Computer-Aided Verification (CAV)*, volume 6806 of *Lecture Notes in Computer Science*, pages 686–702. Springer-Verlag, 2011.

[104] S. A. Seshia, S. Hu, W. Li, and Q. Zhu. Design automation of cyber-physical systems: Challenges, advances, and opportunities. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(9):1421–1434, 2016.

[105] V. Sjöberg and S. Weirich. Programming up to congruence. In *ACM SIGPLAN Notices*, volume 50, pages 369–382. ACM, 2015.

[106] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *International Conference on Functional Programming (ICFP), ACM SIGPLAN Notices*, volume 46, pages 266–278. ACM, 2011.

[107] N. Swamy, J. Weinberger, C. Schlesinger, J. Chen, and B. Livshits. Verifying higher-order programs with the Dijkstra monad. In *Programming Language Design and Implementation (PLDI), ACM SIGPLAN Notices*, volume 48, pages 387–398. ACM, 2013.

[108] M. Tekriwal, K. Duraisamy, and J.-B. Jeannin. A formal proof of the lax equivalence theorem for finite difference schemes. In A. Dutle, M. M. Moscato, L. Titolo, C. A. Muñoz, and I. Perez, editors, *NASA Formal Methods*, pages 322–339, Cham, 2021. Springer International Publishing.

[109] M. Tekriwal, J. Miller, and J.-B. Jeannin. Formal verification of iterative convergence of numerical algorithms, 2022.

[110] A. Tiwari and G. Khanna. Series of abstractions for hybrid automata. In *International Workshop on Hybrid Systems: Computation and Control*, pages 465–478. Springer, 2002.

[111] C. Tomlin, G. J. Pappas, and S. Sastry. Conflict resolution for air traffic management: A study in multiagent hybrid systems. *Automatic Control, IEEE Transactions on*, 43(4):509–521, 1998.

[112] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Refinement types for haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, pages 269–282, 2014.

[113] C. von Essen and D. Giannakopoulou. Analyzing the next generation airborne collision avoidance system. In *TACAS*, volume 8413 of *LNCS*, pages 620–635. Springer, 2014.

[114] T. Wang, R. Jobredeaux, H. Herencia, P.-L. Garoche, A. Dieumegard, É. Féron, and M. Pantel. From design to implementation: an automated, credible autocoding chain for control systems. In *Advances in Control System Technology for Aerospace Applications*, pages 137–180. Springer, 2016.

[115] T. Wang, R. Jobredeaux, M. Pantel, P.-L. Garoche, E. Feron, and D. Henrion. Credible autocoding of convex optimization algorithms. *Optimization and Engineering*, 17(4):781–812, 2016.

[116] T. E. Wang, A. E. Ashari, R. J. Jobredeaux, and E. M. Feron. Credible autocoding of fault detection observers. In *2014 American Control Conference*, pages 672–677. IEEE, 2014.

[117] S. Wolfram. The Mathematica book. *Assembly Automation*, 1999.

[118] T. Wongpiromsarn, S. Mitra, A. Lamperski, and R. M. Murray. Verification of periodically controlled hybrid systems: Application to an autonomous vehicle. *ACM Transactions on Embedded Computing Systems (TECS)*, 11(S2):53, 2012.

[119] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *ACM SIGPLAN Notices*, volume 33, pages 249–257. ACM, 1998.