CAREER: Verifiable Execution and Simulation of Cyber-Physical Systems

Jean-Baptiste Jeannin, University of Michigan

Overview

Modern cyber-physical systems (CPSs) 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 verification, execution and simulation of CPSs are largely disconnected. As a result, 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 create MARVeLus, a programming language unifying verification, execution and simulation of CPSs, bridging this gap and offering a methodology for end-to-end verification of CPSs.

Intellectual Merit

The difficulty to achieve verification, execution and simulation together stems from multiple fronts: the intricacies of the semantics of languages for CPSs, including machine arithmetic; the complication of modeling and communicating with sensors and actuators; and the challenges in accurately modeling, simulating and proving properties about continuous dynamics, especially when coupled with discrete programs.

To resolve those challenges, we will model CPSs as *hybrid systems*, with both discrete and continuous dynamics. MARVeLus will first be designed as a *synchronous language*, in the tradition of languages such as Lustre, Esterel and Signal. Synchronous languages are built around a synchronous clock and targeted to CPSs and embedded systems. They come with strong runtime and memory guarantees, and are the basis of the industrial tool Scade. By building MARVeLus on a synchronous platform, we will leverage their success based on decades of research, and encourage industry adoption through Scade. Second, MARVeLus will enable verification through *refinement types* and an external Satisfiability Modulo Theories (SMT) solver. We will build a dedicated refinement type system to reason about different properties of hybrid systems, including safety and liveness. Third, we will add ordinary differential equations to our synchronous language, inspired by the recent development of the synchronous language Zélus. We will build refinement typing rules allowing the user to *reason about those differential equations* using explicit solutions and invariants. Finally, we will perform verified simulation by formally *verifying numerical algorithms* approximating differential equations, and formally bounding their errors.

As a result, MARVeLus will be a *synchronous language with differential equations and refinement types*, with a *verified simulation* capability. We will apply and evaluate the design of MARVeLus on a small ground robot and a quadcopter in the laboratory, on the industrial aircraft collision avoidance system ACAS X, and with industry partners at Rockwell/Collins Aerospace and NASA. We will also introduce MARVeLus in a graduate class, and collect feedback from students.

Broader Impacts

Using MARVeLus will empower tomorrow's engineers to design cyber-physical systems with strong, formally verified guarantees. It will allow them to bridge gaps in verification, and ensure that the code that is executed is also the code that was verified. In the long run, my goal is to democratize the use of formal verification in the design of cyber-physical systems. I maintain strong ties with industry, including with Raytheon/Collins Aerospace and NASA, which will facilitate industry evaluation and feedback. I am planning to collaborate with a Detroit-area Title 1 middle school to make the students aware of safety issues in engineering. I have created a new class "AERO 350: Fundamentals of Aerospace Computing", to give undergraduates a firm understanding of the use of computing in aerospace engineering. The class is now mandatory for all undergraduates in aerospace engineering. I have also added a verification module to the graduate class "AERO 552: Aerospace Information Systems". Finally, I am active in the organization of workshops, having organized the 2020 and 2021 editions of the Verification Mentoring Workshop (VMW), and currently organizing the 2022 Programming Languages Summit (MWPLS).

CAREER: Verifiable Execution and Simulation of Cyber-Physical Systems

Jean-Baptiste Jeannin, University of Michigan

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, cars, aircraft and spacecraft, but also in chemical plants or nuclear powerplants, the software interacts directly with its physical environment, thus forming a *cyber-physical system* (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 [5, 4, 3, 1]. 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, simulating and verifying. In particular, many tools support execution and simulation with no or very limited support for formal verification; and many tools designed for verification do 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 in practice. 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 due to bugs in the compiler.

The current state of the art can broadly be classified in three categories: synchronous languages, such as Lustre, Scade, Esterel, Signal, Zélus are languages that are high-level and *executable*; Ptolemy, Simulink, Modelica and others focus on *simulation*; differential dynamic logic, zonotopes and SpaceEx, as well as many model-checking-based tools, focus on *verification*. This CAREER project will design a new programming language called MARVeLus (Method for Automated Refinement Verification of Lustre) that will be the first to provide **a unified approach to executing, simulating and verifying hybrid systems. This approach 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.**

A unified approach is difficult to achieve due to the semantic intricacies of CPS languages and machine arithmetic, and the complication of modeling and simulating the 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 *refinement types*: a synchronous approach inherits from decades of rigorous research on synchronous languages, and enables 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 build MARVeLus on a synchronous language? Synchronous languages such as Lustre [70], Lucid Synchrone [39, 38], Esterel [35] and Signal [100], were developed to safely program embedded systems and cyber-physical systems. 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. Lustre is also the basis of the Scade tool, which is widely used in the aerospace and automobile industry, offering a path to industry adoption of MARVeLus. The integration with continuous dynamics and differential equations [34, 33, 25] has been made recently with the Zélus language – including use in industry in a new version of Scade [48]. Another tool widely used in some

industries is Matlab/Simulink. However, Simulink is closed-source and does not have a widely accepted formal semantics (despite some academic attempts [28, 32]), making it difficult to develop sound proof techniques.

Why use refinement types? Refinement types for verification have matured and shown much success recently, for example in the Liquid Haskell [144] and F* [138] languages (note that F* uses richer dependent types); both languages use off-the-shelf SMT solvers in their backend. I believe that it is time to explore their use for the verification of cyber-physical systems. One advantage of refinement types is that they allow the programmer to annotate the executable code directly, rather than specify properties externally. Moreover, in case a property cannot be proved fully automatically, the programmer can add some refinement annotations to help the solver.

Why is now the right time to do this work? Formal verification has matured significantly over the last few years, especially in application areas such as compilation or distributed systems [102, 97, 30, 75, 149, 104, 72]. For software not interacting with physics, there are now several tools performing verification of executable code [137, 138, 130, 144, 101], often using refinement types or dependent types. Automation has also improved dramatically, with less and less manual interaction needed. Specification and verification of synchronous languages was pioneered by synchronous observers [71], and developed using assume/guarantee contracts [110, 109, 46], and more recently with CoCoSpec [40], and model-checking in Kind 2 [41]. Verification of hybrid systems has become practical with the use of zonotopes [10, 57] and the automatic generation of invariants for continuous dynamics [64]. I believe the time is right to bring the verification of hybrid systems to the language level, to ensure accurate verification of executable code.

Applications. To ensure its usability and adoption, the development of MARVeLus will be evaluated in the laboratory and on some case studies. The first evaluation will consist of ensuring that we can run MARVeLus on small vehicles in the laboratory – a ground robot and a quadcopter – on which we will specify and prove simple properties such as collision avoidance. The second evaluation will be the Next-Generation Airborne Collision Avoidance System ACAS X, an industrial collision avoidance system described in detail in Section 3, on which I have verification experience. 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.

Outcomes and Impact. The MARVeLus language 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 implementation, leaving only two pieces to be trusted: (i) the theorem prover and verifier performing the verification, and (ii) the model of the dynamics of the environment. All other parts will be mechanically verified, enabling the utmost level of trust and verification of safety-critical systems. At a time where safety-critical cyber-physical systems are responsible for expensive equipment and human lives, the systems' designers need powerful tools to guarantee safety. This MARVeLus language will empower tomorrow's engineers to design cyber-physical systems with strong, end-to-end, formally verified guarantees.

Research plan. We will design MARVeLus, a *synchronous* programming language with *differential equations* and *refinement types*. The most important influences will be Liquid Haskell [144], F* [137], and Zélus [34], itself based on Lustre [70]. Further influences, among others, will come from Dafny [101] and differential dynamic logic [121] for the verification part; Ptolemy [44] for the simulation part; and Esterel [35] and Signal [100] 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".** The specific research objectives are summarized below: *(see Section 4)*

1. **Design discrete MARVeLus, a synchronous programming language with refinement types.** Starting from the existing design of the (discrete part of) the synchronous language Zélus [34, 33, 25], we will design a refinement type system in the style of Liquid Haskell [144], that can also express temporal logic properties for safety and liveness. In our implementation, verification will be achieved by discharging proof obligations to a Satisfiability Modulo Theories (SMT) solver, CVC5. *(Thrust 1)*.

2. **Extend MARVeLus with Ordinary Differential Equations (ODEs)**, adding verification capabilities and enhancing the design of Zélus [34, 33, 25]. We will introduce typing rules to prove properties about ODEs, and will provide formal guarantees about simulation races *(Thrust 2)*.
3. **Apply MARVeLus on relevant applications of CPSs:** a small ground robot and a quadcopter in the laboratory, and the ACAS X aircraft collision avoidance system *(Thrust 3)*.

Integrated Education and Outreach Plan. I will perform education and outreach activities to introduce verification to students from middle school to graduate school. I will collaborate with a Detroit-area middle school to organize verification-centric activities. I will continue to perform verification research with undergraduate and graduate students, and will introduce homework using MARVeLus in a graduate class I teach. I will also continue to organize local and mentorship workshops for graduate students. *(see Section 7)*

2 Strength and Experiences of the PI

My career goal is to use verification and programming language techniques to democratize end-to-end verification of cyber-physical systems. 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**. 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. Moreover, I am a Computer Scientist but also a faculty in an Aerospace department with an affiliation in Robotics, which opens up many interesting collaborations.

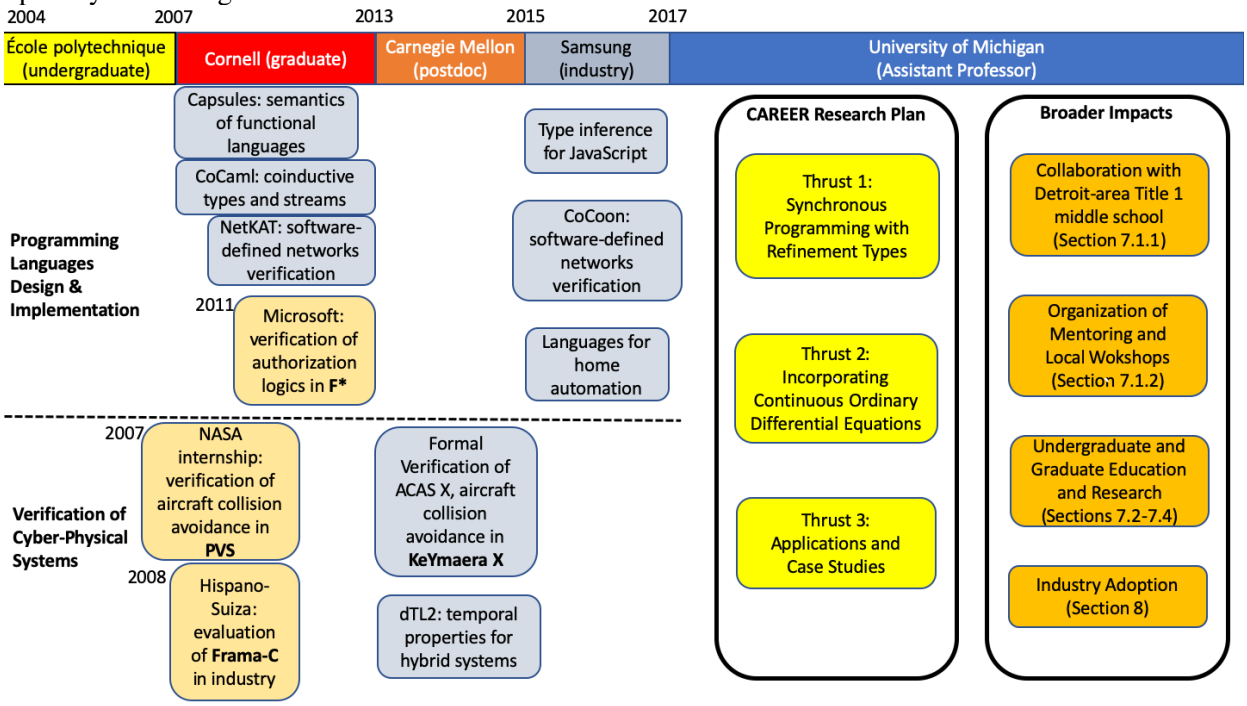

Figure 1: *Overall career path and its relationship with the proposed work plan*

2.1 Verification of cyber-physical systems.

On the theoretical side, I have devised proof systems to incorporate reasoning with linear temporal logic (LTL) and signal temporal logic (STL) formula in a dynamic logic framework [88, 9], and have studied the foundations of quantitative semantics for STL [80]. On the practical side, I have developed an automated method for generating quantifier-free formulations of obstacle avoidance that are efficiently machine-checkable [91].

Case studies. I have performed a verification of the airborne collision avoidance system ACAS X [83, 84, 82] (described in detail in Section 3). With my students, I have also verified the safety of emergency maneuvers for cars [6, 7] and quadcopters [8].

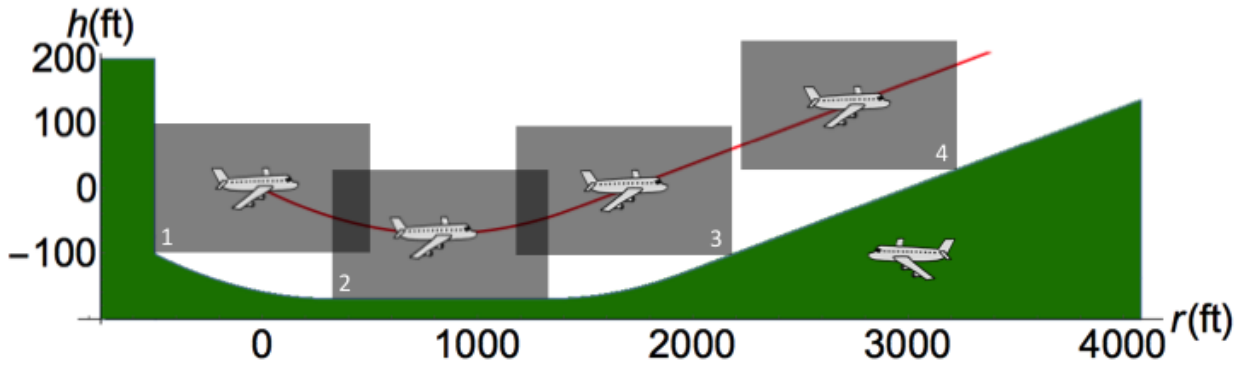

Figure 2: *Safe region for an intruder aircraft (green), for an example "Climb at* 1,500 *feet per minute" advisory*

Verification of numerical methods. With my students, I have been working on verifying the convergence properties of numerical programs, that approximate solutions of differential equations. We have formalized in Coq the Lax–equivalence theorem to prove convergence of a family of finite difference schemes [140], then proved the convergence of solutions obtained from iterative algorithms [141].

2.2 Language Design.

I have extensive experience designing programming languages for different domains, including stream programming. Those past designs have many similar challenges to the design of the MARVeLus language.

Programming with streams 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, by designing the CoCaml language [85, 86, 87].

Software-Defined Networking. Beyond stream programming, I have experience designing languages and verification tools for software-defined networking. I am a co-designer of the NetKAT language [12], which provides a sound and complete axiomatization of a programming language for computer networks, based on Kleene Algebra with Tests (KAT) [95]. I am also a co-designer of the CoCoon language [131], which shows how to build correct by construction, software-defined networks.

3 A motivating example: Formal Verification of Aircraft Collision Avoidance

As a motivating example, let us look at ACAS X, an industrial aircraft collision avoidance system. ACAS X is the Next-Generation Airborne Collision Avoidance System, destined to be installed on airliners as a successor of TCAS (Traffic Collision Avoidance System). Its development is funded by the 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 large table whose domain describes the current state of an encounter, and whose range is a set of scores for each possible action [92, 93]. 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 [92]. 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 [83, 84, 82]. Our approach abstracted away the large table by identifying and proving correct *safe regions* in the state space of the system where we prove formally that a particular advisory, if followed, prevents all possible near-collisions from any aircraft position and velocity in that safe region (Figure 2). The work was successful and uncovered several 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 table. We proved the abstraction of the table with great confidence. But, on the other hand, the 10,500 lines of Julia are quite far from the 170-line KeYmaera X model. **The goal of the MARVeLus project is to close this gap between verified code, executed code and simulation code.**

4 Research Plan

Thrust 1 – Synchronous Programming with Refinement Types

In this first task, we focus strictly on the *discrete* part of our language, and on its verification and execution.

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

In this first task, we will define discrete 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). The syntax of expressions will be built from the discrete fragment of Zélus [34, 33, 25], which is itself inspired from Lucid Synchrone [39, 38], a functional version of Lustre.

$b ::= \mathsf{unit} \mid \mathsf{float} \mid \mathsf{int} \mid \mathsf{bool}$	Base types
$\tau ::= b \mid \tau \times \tau \mid \tau \to \tau$	Simple Types
$\mid \tau\{\phi\}$	Refinement
$e ::= c$	Constants
$\mid x$	Variables
$\mid \mathsf{let}\ x : \tau = e\ \mathsf{in}\ e$	let-binding
$\mid \lambda x.e$	Function
$\mid e\ x$	Application
$\mid (e_1, e_2) \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e$	Pairs
$\mid \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$	Branching
$\mid \mathsf{let\ rec}\ x : \tau = e\ \mathsf{in}\ e$	Recursion
$\mid e : \tau$	Type Annotation
$\mid c\ \mathsf{fby}\ e$	Streams

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 fbyconstructs 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 $(c_1, e_1, e_2, e_3, ...)$. Throughout the development, we will be in constant contact with the Zélus designers *(see letters of collaboration from Marc Pouzet and Timothy Bourke)*.

Preliminary Work. 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 [85, 86, 87]. Along with related work on synchronous languages, this will help us understand the semantics of MARVeLus.

Task 1.2 – Type-refinement Specifications based on Signal Temporal Logic

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

In many cyber-physical systems applications, traditional input/output specifications (e.g., à la Floyd-Hoare Logic) are inadequate. For example, in the case of airplane or car collision avoidance, we want to be able to express that "the vehicles did not collide *throughout* the execution" rather than "the vehicles are properly separated *at the end* of the execution".

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

The state-of-the-art in cyber-physical-systems verification typically uses Signal Temporal Logic (STL) [107]. STL is an extension of Linear Temporal Logic (LTL) where we can express properties about the timing of events. STL is widely used in industry (e.g. at Toyota), and is well-suited to reason about streams and continuous functions. In this task, we will design refinement types using expressions from STL, starting from basic predicates p expressing equalities and inequalities between program expressions. 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. I have designed two extensions of Differential Dynamic Logic, dTL^2 to reason with temporal operators [88], and more recently STd$\mathcal{L}$ to reason with STL specifications [9]. I expect that similar techniques will transfer to reasoning with refinement types. My students and I have devised an initial design for MARVeLus, including types, expressions and trace predicates.

Task 1.3 – Foundations for a Refinement Type System for Discrete MARVeLus

Research Question: How to develop a sound and complete type system for a synchronous language with refinement types inspired from Signal Temporal Logic?

We will define a refinement type system for MARVeLus, along with a refinement-type checker. This allows 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:

$$\frac{\Gamma \vdash c : b\{p\} \qquad \Gamma \vdash e : b\{\Box_{[0,t-1]} p\}}{\Gamma \vdash c \,\mathsf{fby}\, e : b\{\Box_{[0,t]} p\}}$$

Intuitively, this rule says that if refinement p is true in both the initial state of c and everywhere in e up to time $t-1$, then it should be true everywhere in the stream c fby e up to time t.

Completeness. A desirable, foundational result of our refinement type system is its completeness, which we will seek to 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. The typing rules, as well as their soundness and completeness, will be inspired by our previous work on differential temporal dynamic logic [88, 9]. 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, such as the if construct. A possible typing rule for an if statement is the following:

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

To prove that a property ϕ is true for a certain time (i.e., $\Box_{[0,t]}\phi$), a possible rule for streams could be:

$$\frac{\begin{array}{cc}\Gamma \vdash x : \mathsf{bool\ stream} & \Gamma, y : \{\mathsf{int\ stream} : x\} \vdash e_1 : \tau\{\Box_{[0,t]}\phi\}\ \mathsf{stream} \\ y \text{ fresh} & \Gamma, y : \{\mathsf{int\ stream} : \neg x\} \vdash e_2 : \tau\{\Box_{[0,t]}\phi\}\ \mathsf{stream}\end{array}}{\Gamma \vdash \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau\{\Box_{[0,t]}\phi\}\ \mathsf{stream}}$$

However, the premises of this rule are too strong: the rule only allows to conclude if the condition (x, a stream of booleans) is identically true (resp. false) up to time t. But the rule does not allow to conclude anything in the cases where the condition x is true only for some time steps. 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 get something resembling:

$$\frac{\begin{array}{cc}\Gamma \vdash x : \mathsf{bool\ stream} & \Gamma, y : \{\mathsf{int\ stream} : x\} \vdash e_1 : \tau\{\Box_{[0,t]}(x \Rightarrow \phi)\}\ \mathsf{stream} \\ y \text{ fresh} & \Gamma, y : \{\mathsf{int\ stream} : \neg x\} \vdash e_2 : \tau\{\Box_{[0,t]}(x \Rightarrow \phi)\}\ \mathsf{stream}\end{array}}{\Gamma \vdash \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau\{\Box_{[0,t]}\phi\}\ \mathsf{stream}}$$

In this new rule, because the condition x appears under the time quantification (embodied by the $\Box_{[0,t]}$ construct), the time quantification is much stronger, as desired.

Task 1.4 – Implementation

We will implement the features described in the previous tasks, building a prototype of discrete MARVeLus. The implementation will be in OCaml, to leverage some features of the existing Zélus compiler, on which our initial implementation is 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 Scade customers.

Verification conditions and SMT solvers. The verification will leverage existing verification techniques by identifying *proof obligations* in the program and proving them in a state-of-the-art SMT solver [20, 50]. A satisfiability modulo theories (SMT) [24] solver is a solver combining a boolean satisfiability solver (SAT) with different theory solvers (about real numbers, integers, data structures, floating-point, etc.) to determine whether certain mathematical formulae are satisfiable. Identifying proof obligations and discharing to external solvers is a technique also used by e.g. F⋆ [138], Liquid Haskell [144] or Frama-C [49]. We will primarily use the CVC5 SMT solver [20] as a primary back-end. A modern SMT solver with great performance, CVC5 also offers two great advantages for our purpose: it uses a new, efficient variation of cylindrical algebraic decomposition for solving nonlinear real arithmetic as a theory, which will be particularly useful for verifying properties of differential equations [96]; and it supports proof production so that its proofs can be checked by another theorem prover such as Isabelle [21]. Finally, we can count on the CVC5 development team for support and collaboration on the project *(see letter of collaboration from Clark Barrett, one of the main designers of CVC5)*. We will continue to support SMT-LIB-compatible SMT solvers by generating proof obligations compatible with SMT-LIB [23]. 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.

Preliminary Work. My group and I have started implementing verification capabilities in an initial design of MARVeLus. Our initial implementation extends the Zélus compiler, where we have added a compiler pass just after basic type-checking. This pass checks refinement types and generates proof obligations that are currently sent to the Z3 SMT solver [50] (we plan to switch to CVC5.) Our current implementation can currently prove simple examples, such as proving that a discrete version of a thermostat adequately keeps the temperature between desired limits.

Thrust 2 – Incorporating Continuous Ordinary Differential Equations

In this thrust, 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 ordinary differential equations, including continuous invariants, in a refinement type system?

Declaring variables following ODEs. 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 real variables x and y following the differential equations $x' = y$ and $y' = -x$ with initial conditions $x(0) = 1$ and $y(0) = 0$, the programmer will be able to write the declaration below, leading us to extend the possible expressions in the language with the der keyword (referring to the first derivative).

$$\texttt{let der x = y init 1 and der y = -x init 0 in } e \qquad\qquad e ::= \ldots \mid \texttt{let der } x = e \texttt{ init } e \texttt{ and} \ldots \texttt{ in } e \mid \texttt{up}(x)$$

Reasoning about ODEs. We will introduce refinement typing rules to reason about Ordinary Differential Equations (ODEs). These rules will be inspired by the invariant and differential equations proof rules of differential dynamic logic [121, 64]. For example, in the example above, (x,y) describes the circle centered in $(0,0)$ and of radius 1. We would like to be able to prove that $\mathtt{x}^2 + \mathtt{y}^2 = 1$ at all times, and we can do so using an instantiation of a type invariant rule resembling:

$$\frac{\vdash 1^2+0^2=1 \qquad \vdash (x^2+y^2)'=2xx'+2yy'=2xy-2yx=0 \qquad x:\mathtt{real},y:\mathtt{real}\{\Box(x^2+y^2=1)\}\vdash e:\tau}{\vdash \mathtt{let\ der}\ x:\mathtt{real}=y\ \mathtt{init}\ 1\ \mathtt{and\ der}\ y:\mathtt{real}\{\Box(x^2+y^2=1)\}=-x\ \mathtt{init}\ 0\ \mathtt{in}\ e:\tau}$$

Note that, thanks to invariants, we did not need to solve the differential equation using trigonometric functions.

We will focus on ordinary differential equations with time derivatives, with a certain degree of regularity (typically Lipschitz-continuity), ensuring existence, uniqueness and "niceness" of solutions. As a stretch goal, we will consider adding more complicated continuous dynamics such as differential algebraic equations.

Zero-crossings. The interface between the discrete and continuous parts of MARVeLus will be achieved using zero-crossings, inspired from the Zélus language. A zero-crossing $\mathsf{up}(x)$ happens when the solution of a differential equation x crosses zero, spawning a discrete event. We will develop specific refinement typing rules to handle zero-crossings, as they define the interactions between discrete and continuous behaviors.

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$\mathcal{L}$ [122].

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. In MARVeLus, we model both the system and its environment. As such, every variable of the language can be: (i) either discrete or continuous; and (ii) executed by the system or an effect of the environment. 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 physical laws. While this is typical, it is not always the case: proportional-integral controllers are continuous (but often implemented in discrete software), while ball bounces can be modeled as discrete events. We will allow the programmer to specify whether each variable is part of the system execution (then it is compiled) or the environment (then it is not). This will be done as an annotation on the variable type.

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. 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 by 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.) The type system will ensure that real variables cannot be program variables that would be compiled.

Based on my verification experience [84, 140], it is often easier to carry out formal verification by making the simplifying assumption that floating-point numbers behave as if they were real numbers. Many verification conditions become simplified (e.g., overflows do not have to be dealt with), making the verification task more tractable. In MARVeLus, we will introduce a compilation flag conflating reals and floats for verification

purposes. Of course, this simplification comes at a cost: if an unexpected event such as a floating-point overflow happens, then the guarantees of the verification are not longer valid. A similar distinction exists between (mathematical) natural numbers and machine integers. Mathematical natural numbers cannot overflow, and are e.g. represented by the type `nat` in Coq. 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 desired.

Preliminary Work. Although I 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 [67]. 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

Research Question: How to provide guarantees on the simulation of a cyber-physical system?

The simulation of dynamical systems that use ODEs typically rely on unverified, off-the-shelf numerical algorithms to approximate the solutions of those ODEs. For example, the Zélus tool uses Sundials CVODE [47]. Because those tools are unverified, there is no guarantee that the numerical solution of the differential equation is correct within a known error bound, and some bugs are known in e.g. Simulink [26]. Adapting and extending recent work on formally verified numerical methods [140, 141], we will develop tools to obtain formal guarantees on the approximated solutions of differential equations. We will be able to guarantee that the simulated traces are actually traces of the model, within a known, formally verified error bound. Even if we do recognize that a simulation can only be as good as its model, we believe that there is value in guaranteeing that a trace is a valid trace of a model, and in bounding its numerical error.

Integration with Verified Compilation. As a stretch goal, we will integrate our formally verified simulation of differential equations with an existing formally verified compilation framework for Lustre [30, 31]. We will work with the authors of the framework, and this will provide a full, discrete-and-continuous, verified simulation of cyber-physical systems *(see letters of collaboration from Marc Pouzet and Timothy Bourke)*. The formal guarantee we will provide is that the generated simulation trace is actually a trace of the model, within a known error bound.

Preliminary Work. We have already formalized certain results pertaining to numerical methods for finite difference schemes [140], as well as iterative methods for approximating the results of linear systems [141].

Task 2.4 – Implementation

We will extend discrete MARVeLus to handle differential equations following the extended refinement type system. We will also generate proof obligations that will then be sent to the CVC5 SMT solver. For this part, we will particularly need the benefits of the nonlinear arithmetic solver of CVC5 [96].

Thrust 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 kinds of applications: a ground robot and a quadcopter in the lab; an industrial application for ACAS X aircraft collision avoidance; and as homework in a graduate class with a verification module.

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

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

We will run and evaluate MARVeLus on a small differential-drive wheeled robot, the M-Bot Mini (Figure 3). M-Bot Mini is equipped with on-board sensors including a 9-axis inertial measurement unit (IMU), a rotating LiDAR, and a front-facing optical camera. The drivetrain consists of two geared DC motors. The robot is


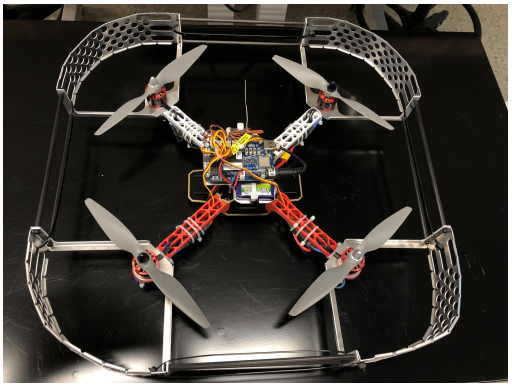

Figure 3: *The M-Bot Mini Ground Robot and the M-330 Quadcopter, used in my laboratory for running MARVeLus on hardware*

equipped with a Beaglebone Blue, a Linux-capable single-board computer featuring built-in motor controllers. We will focus on verifying collision avoidance properties. Running MARVeLus on real hardware will give us crucial insights into the design of the language, and as to the limits of the verification guarantees.

Preliminary Work. We already own and operate two M-Bots Mini in my laboratory, and we are able to run simple MARVeLus code (move forward and backward, simple turns). The MARVeLus runtime can be configured to run remotely on a separate computer, or locally on-board the robot's Beaglebone Blue. 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. In both configurations, the MARVeLus runtime communicates with the controller code via the Lightweight Communications and Marshalling (LCM) protocol.

Task 3.2: Flying robot: the M-330 Quadcopter

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

We will also run and evaluate MARVeLus on the M-330 Quadcopter (Figure 3), first focusing on quadcopter collision avoidance properties. 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.

Preliminary Work. We already operate am M-330 Quadrotor in the laboratory. Our recent paper on quadcopter collision avoidance [8] provides safety regions, 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

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

In this task, we will close the gap between verification, execution and simulation on ACAS X, the initial motiva-

tion for the MARVeLus project. Thanks to my past verification work on ACAS X, I have a deep understanding of the intricacies of the system and its verification challenges. We will build a new implementation of ACAS X using MARVeLus, equip it with the differential equations modeling simplified flight dynamics, and perform an end-to-end verification of ACAS X in MARVeLus. As a stretch goal, we will aim to make our verified implementation an alternate reference implementation of ACAS X (the current reference implementation is unverified and in Julia). Throughout this case study, I will leverage past interactions with different actors of the ACAS X project, including with the Federal Aviation Administration and the main designers of ACAS X.

Preliminary Work. Beyond our verification work of a hybrid-systems model of ACAS X [83, 84, 82], we have recently introduced an automation technique using active corners that will simplify the proof [91].

Task 3.4: Incorporation in AERO 552 Aerospace Information Systems

Research Question: How to ensure graduate students and engineers can use MARVeLus effectively?

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 lacks hands-on components. Once a stable prototype of MARVeLus is built, I will incorporate the use of MARVeLus in two homeworks, so that students can understand how to use verification tools in practice. The case study will be a simplified version of our ACAS X work, where we envision using very simplified dynamics. 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 useful for the development and evaluation of MARVeLus.

5 Brief Overview of Related Research

5.1 Language and Tools for Cyber-Physical Systems

Synchronous languages such as Lustre/SCADE [70], Esterel [35] and Signal [100] 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. [30, 31] formally verified a Lustre compiler in the Coq proof assistant. In general, cyber-physical systems are notoriously difficult to design correctly [134].

The Ptolemy and Ptolemy II projects [51, 126] 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 exceptions using model-checking [44] or the Real Time Maude tool [117, 16]. Simulink [118] and Modelica [59] can simulate complete systems, including a continuous environment, and Simulink offers code generation without correctness guarantees. The Zélus language [34] 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. [14] and Alur et al. [11] created a basis to reason about hybrid systems. Model checking has been adapted to reasoning about hybrid sytems and hybrid automata [76]. Differential Dynamic Logic [121] extends Dynamic Logic [73] with differential equations to formally verify important safety and liveness properties of hybrid systems. SpaceEx [57] and CORA [10] explore reachability properties of hybrid systems with numerical approximations and zonotopes, enabling scalability. S-Taliro [13] and Breach [52] can find counter-example in Matlab/Simulink models. Kind 2 [41, 69] enables k-induction model-checking for the synchronous language Lustre. CoCoSpec [40] and CoCoSim [29] focus on verification of Simulink models, while UCLID5 integrates modeling and verification [135]. There are many approaches to generating invariants for cyber-physical systems, e.g., [63, 64, 133, 42, 43, 132, 129, 142, 68], and for synthesis of control algorithms [127, 128, 116].

The idea of languages that permit both verification and execution is not new. Koord [65] and CyPhyHouse [66] focus on high-level programming for CPSs. VeriDrone [106] proves drone properties in the Coq

theorem prover. VeriPhy [27] presents an end-to-end verification of models verified in Differential Dynamic Logic, at a low-level. Autocoding [89, 148, 147, 146] 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 [105] language focuses on testing of autonomous driving systems, the PGCD language [19] on robot programming and verification, and the Scenic language [58] on scenario specification and scene specification for testing machine learning algorithms, especially vision. HyST [17] is a source transformation tool for hybrid automaton models, but does not provide a high level of abstraction to build large systems. Wongpiromsarn et al. [150] provide new conditions for periodic hybrid automata.

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 [124, 125, 108] and model checking [45, 18, 77].

The seminal works of Floyd [56] and Hoare [79] on assigning meanings to programs has been extended into dynamic logic [55, 73], the μ-calculus [94], and Kleene algebra with tests [12, 95]. Modern tools such as Dafny [101], Frama-C [49] and Krakatoa [54] use variants of these techniques at their heart. Dependent types [153, 15, 152] is an alternate approach to program verification, and its modern incarnations include he F* project [137, 138, 139], Dependent ML [151], Agda [36], and Zombie [37, 136]. Interactive theorem provers allow users to provide formal proofs of mathematical theorems, including proofs of programs. They include Coq [22], Lean [111], Isabelle [120], HOL and HOL Light [74], PVS [119] and others. Finally, the logic TLA [98] 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 [60, 84, 84] during my postdoctoral studies, the F* language [137, 81] at Microsoft Research, the PVS theorem prover [119, 78] at NASA, the Frama-C framework [49] in industry at Hispano-Suiza, as well as the Coq theorem prover [22, 140] and dynamic logic in recent projects [6, 7] (Fig. 1).

5.3 Aircraft Collision Avoidance

Many efforts have explored developing correct and comprehensive guarantees about collision avoidance decisions for aircraft and other vehicles. Collision avoidance algorithms are developed for both horizontal and vertical motion in 2D and 3D, and formally verified with PVS [113, 114, 53, 61]. The logic for the Traffic Collision Avoidance System (TCAS) is formalized in PVS and used geometries that generate advisories [112]. For the ACAS X system, the state space of a similar MDP using probabilistic model checking and an adaptive Monte Carlo tree search was analyzed [145, 99]. [115] 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 [62, 103, 123, 143].

6 Implementation Plan and Timeline

Table 1 shows a timeline of our proposed work. The project will support one Ph.D. student for five years, and undergraduates will participate where appropriate. I will support undergraduates in the project through dedicated programs at the University of Michigan (e.g., REUs, SURE and AURA programs).

7 Broader Impacts

7.1 Integrated Education and Outreach Plan

I believe it is important to introduce students as early as possible to safety and verification in engineering. For this reason, my education plan targets students at the middle school level — where students see engineeering for the first time — as well as at the undergraduate and graduate levels. I will collaborate with a Detroit-area middle

#	**Thrust** / Task	**Yr 1**	**Yr 2**	**Yr 3**	**Yr 4**	**Yr 5**
	Thrust 1: Synchronous Programming with Refinement Types					
1.1	Discrete MARVeLus: Syntax, Simple Type System					
1.2	Type-refinement Specifications based on SLT					
1.3	Foundations of a Refinement Type System					
1.4	Implementation					
	Thrust 2: Incorporating Continuous Ordinary Differential Equations					
2.1	A Refinement Type System for Differential Equations					
2.2	Compilation Modes for Simulation and Execution					
2.3	Formally Verified Simulation					
2.4	Implementation					
	Thrust 3: Applications and Case Studies					
3.1	Small Ground Robot: the M-Bot Mini Ground Robot					
3.2	Flying Robot: the M-330 Quadcopter					
3.3	Aircraft Collision Avoidance and ACAS X					
3.4	Incorporation in AERO 552 Information Systems					

Table 1: *Proposed project timeline. Each task is marked with a box for each quarter that we expect it to be active. Quarters marked with a "■" indicate a tentative software release in that quarter.*

school, and introduce verification through undergraduate and graduate classes, research, and workshops.

7.1.1 Collaboration with Title 1 middle school in Detroit suburb

I will collaborate with Samantha Coram and Sabrina Cristofaro, science teachers at Ferndale Middle School, a **Title 1 school in Ferndale, Michigan**, a suburb of Detroit *(see collaboration letters)*. The school currently has 499 students enrolled, including 56% Black or African American, 31% White, and 13% from other races. 76% of students qualify for free or reduced lunch. Ferndale middle school is also a School of Choice, attracting students from every part of Ferndale, but also from the neighboring cities of Oak Park and Hazel Park, also suburbs of Detroit. The school is a 45-minute drive from the campus of the University of Michigan.

Samantha Coram and Sabrina Cristofaro are currently teaching science following the Project Lead The Way [2] curriculum in 7th and 8th grade. The classes have about 35 students per semester, hence we expect to reach a total of **350 students over ten semesters**. The class meets 5 days a week throughout the semester. We will collaborate in the "Design and Modeling" class to enhance the student's awareness with safety of critical systems, and familiarize the students with the concept of verification. Projects in their class focus on providing tools for children with cerebral palsy, a congenital disorder marked by impaired muscle coordination.

During each semester of the five years of the project (ten semesters total), my Ph.D. student and I will come to Ferndale middle school once a week for four weeks, for each semester ($2,000 of the budget was allocated to travel to the school). I will first give **a presentation regarding safety and verification**, targeted to middle schoolers, followed by **some hands-on experience** on a simple prosthetics project for children with cerebral palsy. The presentation will show the students how to use simple geometric arguments from their mathematics class to develop conditions ensuring that the prosthetic does not harm the patient by e.g. overbending some muscles, or moving too fast or with too much force. 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 (on a mannequin) after 3D printing the prosthetics. 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 systems**, and the realization that scientific tools exist to ensure this safety. $5,000 of the budget is allocated to the purchase of 3D printers and robotics motors, which will be essential to the successful

completion of this project, as it will allow students to test their concept in physical life.

Long term, Samantha Coram, Sabrina Cristofaro and I will establish new orientations and objectives of the class. We 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 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 prosthetics, delivery drones and self-driving cars of tomorrow.

7.1.2 Organization of Workshops

Organization of Verification Mentoring Workshops. I have a history of organizing mentoring workshops, especially targeted to underrepresented minorities. I was an organizer in 2020, and a co-chair in 2021, of the Verification Mentoring Workshops (VMW), which each attracted more than 400 registrants, and over 50 participants at each session. VMW is a workshop for students held the first two days of the Computer-Aided Verification (CAV) conference, whose purpose is to provide mentoring and career advice to early-stage graduate students and late-stage undergraduate students. The workshop particularly encourages participation of women and underrepresented minorities. Due to the pandemic, both workshops were held online, an opportunity to serve many more participants. In 2020, 22% of the registrants identified as women, 0.7% as nonbinary, 1.4% as Black, African or African-American, and 0.7% identified as Hispanic. Long term, I am planning to stay involved in the organization of mentoring workshops, including both VMW and PLMW.

Organization of the Midwest Programming Languages Summit (MWPLS). I am organizing the 2022 MWPLS, a regional wokshop offering a friendly venue for graduate students of all backgrounds to present in-progress work. We particularly encourage presentations from women and students from underrepresented minorities, and students who are still early in their studies.

7.2 Undergraduate Research

I am dedicated to broadening participation in engineering education. I have hosted over fifteen undergraduate researchers in my group, including 3 women, and I regularly recruit through the Summer Undergraduate Research in Engineering (SURE) program. Five of the mentored undergraduates have written papers [7, 6, 8, 90, 91, 80], one went on to a Ph.D. program and another to a masters program. Over the last three summers, I have also hosted four Ethiopian students from the Addis Ababa Institute of Technology (AAiT), through the African Undergraduate Research Adventure (AURA) program. One of them is now a Ph.D. student in the US, and two others are applying for Ph.D. programs.

7.3 Graduate Education and Research

I teach the graduate course AERO 552: Aerospace Information Systems, where I will introduce homeworks using MARVeLus (see Task 3.4). I already offered this graduate course four times in Fall 2017, Winter 2019, Winter 2021 and Winter 2022. Most of the funds requested for this project will be used to support a Ph.D. student. I currently advise or co-advise four graduate students: one in Aerospace Engineering, one in Computer Science and Engineering, and two in Robotics. I will primarily recruit students from those programs.

7.4 Undergraduate Education

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, including fundamentals of programming, computational science and embedded systems and an introduction to verification. I have taught the class 4 times (Fall 2018, 2019, 2020, 2021), and it has been a success with an average score of 4.2/5.0. The class is now mandatory for all undergraduates in Aerospace Engineering.

7.5 Dissemination via Publications and Open-Source software

The results of this multidisciplinary project will be published in the venues four different communities: programming languages (POPL, PLDI, SPLASH), cyber-physical systems (HSCC, ICCPS), verification (CAV, TACAS, IJCAR), and embedded software (EMSOFT). Starting the third year, I will organize tutorials at the HSCC and SPLASH conferences to bring together the programming languages and cyber-physical systems

communities. I will especially encourage participation from my industry collaborators. The software from this research program will be released under the modified BSD license, and disseminated through GitHub.

8 Evaluation Plan

The main evaluation tool for our research plan on MARVeLus will be the different tasks of Thrust 3: experiments on small robots in the laboratory (tasks 3.1 and 3.2), programming of ACAS X (tasks 3.3), and use of the MARVeLus language with graduate students (task 3.4). By implementing physical systems and an industrial collision avoidance system, we will gain invaluable feedback on the design of MARVeLus. Furthermore, introducing MARVeLus in a classroom will show us first-hand what novice users struggle with. We will provide survey questions to the students further our evaluation, and compare the answers over the years.

Industry feedback and adoption. An essential part of our evaluation will be the technology transfer to industry. I have established collaborations with the Raytheon/Collins Aerospace, the National Aeronautics and Space Administration (NASA) and the Toyota Research Institute, through past or present funding provided by those entities *(see letters of collaboration from Ebad Jahangir at Raytheon and Aaron Dutle at NASA)*. I will present our work on MARVeLus to our different industrial partners, and will encourage them to try it and give us feedback. I will also invite them to workshops and tutorials on MARVeLus that we will organize. I recognize that getting industry traction is a difficult task, but I hope to be successful thanks to our continuing collaborations and the compatibility of MARVeLus with existing industrial languages such as Scade.

Evaluation of education plan. I will use several tools to evaluate the impact of my educational plan. For the middle school collaboration, I will talk directly to the students and perform some simple surveys (subject to approval). For the workshops, I will also rely on surveys sent to the participants. I will evaluate new homework in the AERO 552 class using the official student evaluations of the class, as well as an assessment by the Center for Research on Learning and Teaching (CRLT) at the University of Michigan. Finally, I will evaluate my undergraduate research by individual conversations and assessing the number of students who become excited about formal verification and want to engage in research projects.

9 Risks and Mitigations

We recognize that designing a language such as MARVeLus is a major undertaking, and we do not take the difficulty of the task lightly. During our preliminary work on designing a refinement type system for a (discrete) stream calculus, we have already encountered some difficult rule designs. We see the design of the applications in the laboratory (tasks 3.1 and 3.2) as decisive to validate the design of MARVeLus. A risk is that MARVeLus will prove inadequate to run on the laboratory robots. To mitigate this risk, we will perform the language design and the applications hand-in-hand, so that we can get early feedback on parts that might cause problems.

A major risk that we identify is that the MARVeLus language becomes difficult to use for new users, hurting adoption by students and industry. This is common in formal verification, where many languages come with steep learning curves. To mitigate this risk, we will constantly solicit feedback from students and industry on our design choices. We will be ready to respond to feedback as needed, and change the design as needed.

10 Results from Prior 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 are widely used in science and engineering. This simulation process involves errors and uncertainties, that we seek to quantify and formally prove correct. 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. **Broader Impacts.** The proposed activities – centered around formal verification – have the potential to re-invigorate existing links between theoretical computer science and computational science. 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.

References

[1] Boeing Starliner's Crew Capsule Fails to Enter Planned Target Orbit for Space Station Docking. https://techcrunch.com/2019/12/20/boeings-starliner-crew-capsule-fails-to-enter-planned-target-orbit-for-space-station-docking. Accessed: 2022-07-20.

[2] Project Lead The Way. https://www.pltw.org/. Accessed: 2022-07-20.

[3] The Worst Computer Bugs in History: Rapid unanticipated disassembly of the Mars Climate Orbiter. https://www.bugsnag.com/blog/bug-day-mars-climate-orbiter. Accessed: 2022-07-20.

[4] The Worst Computer Bugs in History: The Ariane 5 Disaster. https://www.bugsnag.com/blog/bug-day-ariane-5-disaster. Accessed: 2022-07-20.

[5] Therac-25 causes radiation overdoses. https://ethicsunwrapped.utexas.edu/case-study/therac-25. Accessed: 2022-07-20.

[6] 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 (HSCC)*, pages 1–11, 2020.

[7] 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.

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

[9] H. Ahmad and J.-B. Jeannin. A program logic to verify signal temporal logic specifications of hybrid systems. In *Proceedings of the 24th International Conference on Hybrid Systems: Computation and Control*, pages 1–11, 2021.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, and A. Ozdemir. cvc5: a versatile and industrial-strength smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 415–442. Springer, 2022.

[21] H. Barbosa, A. Reynolds, G. Kremer, H. Lachnitt, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Viswanathan, S. Viteri, Y. Zohar, C. Tinelli, and C. Barrett. Flexible proof production in an industrial-strength smt solver. In *International Joint Conference on Automated Reasoning (IJCAR)*, 2022, to appear.

[22] 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.

[23] C. Barrett, A. Stump, and C. Tinelli. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, volume 13, page 14, 2010.

[24] C. Barrett and C. Tinelli. Satisfiability modulo theories. In *Handbook of model checking*, pages 305–343. Springer, 2018.

[25] A. Benveniste, T. Bourke, B. Caillaud, J.-L. Colaço, C. Pasteur, and M. Pouzet. Building a hybrid systems modeler on synchronous languages principles. *Proceedings of the IEEE*, 106(9):1568–1592, 2018.

[26] A. Benveniste, T. Bourke, B. Caillaud, B. Pagano, and M. Pouzet. A type-based analysis of causality loops in hybrid systems modelers. *Nonlinear Analysis: Hybrid Systems*, 26:168–189, 2017.

[27] 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.

[28] O. Bouissou and A. Chapoutot. An operational semantics for simulink's simulation engine. *ACM SIGPLAN Notices*, 47(5):129–138, 2012.

[29] H. Bourbouh, G. Brat, and P.-L. Garoche. Cocosim: an automated analysis framework for simulink/stateflow. In *Model Based Space Systems and Software Engineering-European Space Agency Workshop (MBSE 2020)*, 2020.

[30] 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.

[31] T. Bourke, L. Brun, and M. Pouzet. Mechanized semantics and verified compilation for a dataflow synchronous language with reset. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–29, 2019.

[32] T. Bourke, F. Carcenac, J.-L. Colaço, B. Pagano, C. Pasteur, and M. Pouzet. A synchronous look at the simulink standard library. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):176, 2017.

[33] T. Bourke, J.-L. Colaço, B. Pagano, C. Pasteur, and M. Pouzet. A synchronous-based code generator for explicit hybrid systems languages. In *International Conference on Compiler Construction*, pages 69–88. Springer, 2015.

[34] 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.

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

[36] 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.

[37] 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.

[38] P. Caspi, G. Hamon, and M. Pouzet. Synchronous functional programming: The lucid synchrone experiment. *Real-Time Systems: Description and Verification Techniques: Theory and Tools. Hermes*, pages 28–41, 2008.

[39] P. Caspi and M. Pouzet. A functional extension to lustre. *Intensional Programming I*, page 15, 1995.

[40] A. Champion, A. Gurfinkel, T. Kahsai, and C. Tinelli. Cocospec: A mode-aware contract language for reactive systems. In *International Conference on Software Engineering and Formal Methods*, pages 347–366. Springer, 2016.

[41] A. Champion, A. Mebsout, C. Sticksel, and C. Tinelli. The kind 2 model checker. In *International Conference on Computer Aided Verification*, pages 510–517. Springer, 2016.

[42] 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.

[43] 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.

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

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

[46] A. Cohen, L. Gérard, and M. Pouzet. Programming parallelism with futures in lustre. In *Proceedings of the tenth ACM international conference on Embedded software*, pages 197–206, 2012.

[47] S. D. Cohen, A. C. Hindmarsh, and P. F. Dubois. Cvode, a stiff/nonstiff ode solver in c. *Computers in physics*, 10(2):138–143, 1996.

[48] J.-L. Colaço, B. Pagano, and M. Pouzet. Scade 6: A formal language for embedded critical software development. In *2017 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 1–11. IEEE, 2017.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

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

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

[57] 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.

[58] 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.

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

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] 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.

[65] R. Ghosh, C. Hsieh, S. Misailovic, and S. Mitra. Koord: a language for programming and verifying distributed robotics application. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 2020.

[66] R. Ghosh, J. P. Jansch-Porto, C. Hsieh, A. Gosse, M. Jiang, H. Taylor, P. Du, S. Mitra, and G. Dullerud. Cyphyhouse: A programming, simulation, and deployment toolchain for heterogeneous distributed coordination. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6654–6660. IEEE, 2020.

[67] 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.

[68] 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.

[69] G. Hagen and C. Tinelli. Scaling up the formal verification of lustre programs with smt-based techniques. In *2008 Formal Methods in Computer-Aided Design*, pages 1–9. IEEE, 2008.

[70] 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.

[71] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In *Algebraic Methodology and Software Technology (AMAST?93)*, pages 83–96. Springer, 1994.

[72] T. Hance, M. Heule, R. Martins, and B. Parno. Finding invariants of distributed systems: It's a small (enough) world after all. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 115–131, 2021.

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

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

[75] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. Ironfleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 1–17, 2015.

[76] 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.

[77] 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.

[78] 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.

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

[80] J.-B. Jeannin, J. Chen, J. Vargas de Mendonça, and K. Mamouras. Work-in-progress: Towards a theory of robust quantitative semantics for signal temporal logic. In *International Conference on Embedded Software (EMSOFT)*, 2022, to appear.

[81] 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.

[82] 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.

[83] 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.

[84] 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.

[85] 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.

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

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

[88] 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.

[89] 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.

[90] 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.

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

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

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

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

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

[96] G. Kremer, A. Reynolds, C. Barrett, and C. Tinelli. Cooperating techniques for solving nonlinear real arithmetic in the cvc5 smt solver. In *International Joint Conference on Automated Reasoning (IJCAR)*, 2022, to appear.

[97] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: a verified implementation of ML. In *Principles of Programming Languages (POPL), ACM SIGPLAN Notices*, volume 49, pages 179–191. ACM, 2014.

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

[99] 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.

[100] 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.

[101] 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.

[102] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363, 2009.

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

[104] H. Ma, A. Goel, J.-B. Jeannin, M. Kapritsos, B. Kasikci, and K. A. Sakallah. I4: incremental inference of inductive invariants for verification of distributed protocols. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 370–384, 2019.

[105] 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.

[106] G. Malecha, D. Ricketts, M. M. Alvarez, and S. Lerner. Towards foundational verification of cyber-physical systems. In *2016 Science of Security for Cyber-Physical Systems Workshop (SOSCYPS)*, pages 1–5. IEEE, 2016.

[107] 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.

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

[109] F. Maraninchi and L. Morel. Arrays and contracts for the specification and analysis of regular systems. In *Proceedings. Fourth International Conference on Application of Concurrency to System Design, 2004. ACSD 2004.*, pages 57–66. IEEE, 2004.

[110] F. Maraninchi and L. Morel. Logical-time contracts for reactive embedded components. In *Proceedings. 30th Euromicro Conference, 2004.*, pages 48–55. IEEE, 2004.

[111] L. d. Moura, S. Kong, J. Avigad, F. v. Doorn, and J. v. Raumer. The lean theorem prover (system description). In *International Conference on Automated Deduction*, pages 378–388. Springer, 2015.

[112] 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.

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

[114] 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.

[115] 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.

[116] 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.

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

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

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

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

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

[122] 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.

[123] 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.

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

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

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

[127] 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.

[128] 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.

[129] 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.

[130] P. M. Rondon, M. Kawaguci, and R. Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 159–169, 2008.

[131] 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.

[132] 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.

[133] 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.

[134] 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.

[135] S. A. Seshia and P. Subramanyan. Uclid5: Integrating modeling, verification, synthesis and learning. In *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, pages 1–10. IEEE, 2018.

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

[137] 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.

[138] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Zanella-Bguelin. Dependent types and multi-monadic effects in F*. In *Principles of Programming Languages (POPL), ACM SIGPLAN Notices*, volume 51, pages 256–270. ACM, 2016.

[139] 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.

[140] 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.

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

[142] 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.

[143] 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.

[144] 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.

[145] 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.

[146] 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.

[147] 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.

[148] 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.

[149] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 357–368, 2015.

[150] 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.

[151] H. Xi. Dependent ml an approach to practical programming with dependent types. *Journal of Functional Programming*, 17(2):215–286, 2007.

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

[153] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 214–227. ACM, 1999.