Synchronous Programming and Refinement Types in Robotics: From Verification to Implementation

Jiawei Chen*
chenjw@umich.edu
University of Michigan
Ann Arbor, MI, USA

José Luiz Vargas de Mendonça
joselvdm@umich.edu
University of Michigan
Ann Arbor, MI, USA

Shayan Jalili
sjalili@umich.edu
University of Michigan
Ann Arbor, MI, USA

Bereket Ayele
bereketshimels77@gmail.com
Addis Ababa Institute of Technology
Addis Ababa, Ethiopia

Bereket Ngussie Bekele
bereketngussie99@gmail.com
Addis Ababa Institute of Technology
Addis Ababa, Ethiopia

Zhemin Qu
quzhemin@umich.edu
University of Michigan
Ann Arbor, MI, USA

Pranjal Sharma
spranjal@umich.edu
University of Michigan
Ann Arbor, MI, USA

Tigist Shiferaw
tgg909441@gmail.com
Addis Ababa Institute of Technology
Addis Ababa, Ethiopia

Yicheng Zhang
zyicheng@umich.edu
University of Michigan
Ann Arbor, MI, USA

Jean-Baptiste Jeannin
jeannin@umich.edu
University of Michigan
Ann Arbor, MI, USA

Abstract

Robots and other cyber-physical systems are held to high standards of safety and reliability, and thus one must be confident in the correctness of their software. Formal verification can provide such confidence, but programming languages that lend themselves well to verification often do not produce executable code, and languages that are executable do not typically have precise enough formal semantics. We present MARVeLus, a stream-based approach to combining verification and execution in a synchronous programming language that allows formal guarantees to be made about implementation-level source code. We then demonstrate the end-to-end process of developing a safe robotics application, from modeling and verification to implementation and execution.

*Corresponding Author

FTSCS '22, December 07, 2022, Auckland, New Zealand

ACM ISBN 978-1-4503-9907-4/22/12…$15.00
https://doi.org/10.1145/3563822.3568015

CCS Concepts: • **Computer systems organization** → **Robotics**; **Embedded systems**; • **Software and its engineering** → **Formal software verification**; **Data flow languages**; • **Theory of computation** → **Type theory**.

Keywords: formal verification, synchronous programming, refinement types, cyber-physical systems, robotics

ACM Reference Format:
Jiawei Chen, José Luiz Vargas de Mendonça, Shayan Jalili, Bereket Ayele, Bereket Ngussie Bekele, Zhemin Qu, Pranjal Sharma, Tigist Shiferaw, Yicheng Zhang, and Jean-Baptiste Jeannin. 2022. Synchronous Programming and Refinement Types in Robotics: From Verification to Implementation. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Formal Techniques for Safety-Critical Systems (FTSCS '22), December 07, 2022, Auckland, New Zealand.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3563822.3568015

1 Introduction

Cyber-Physical Systems (CPS) are becoming increasingly ubiquitous in modern life, and the rise of automation has led to ever more exacting standards of reliability. From industrial controls and avionics to robots working alongside people, many CPS have the potential to cause significant harm or injury if improperly designed. However, their complex interactions with the environment make these some of the most challenging systems to safely design, as testing is often not thorough enough to cover all possible edge cases. For this reason, formal verification is an attractive approach to ensuring CPS safety, as it is able to provide rigorous guarantees on safety given proper assumptions on the system. However,

the thoroughness of formal verification often comes at the cost of expressivity. Languages designed for formal verification may be too abstract to accurately capture the behavior of real systems [1, 41]. As a result, verification may involve translation, which may abstract away potential bugs in the original system or expand the trusted code base. [4, 11, 36]. On the other hand, tools that produce executable CPS often do not have precise enough semantics for formal verification. Thus, a single language that strives to enable CPS verification and execution must strike a balance between precision for the former and expressivity for the latter.

We present a new language MARVeLus (**M**ethod for **A**utomated **R**efinement-type **Ve**rification of **Lus**tre), an extension of the synchronous programming language Zélus [8] designed with this goal in mind, primarily targeting verified robotics applications. MARVeLus combines the expressivity of synchronous programming in a language like Lustre [16], with the formal verification offered by a refinement type system, such as that found in Liquid Haskell [45]. To our knowledge, MARVeLus is unique in applying this to execution of programs on real hardware once they have been verified, allowing for a unified specification, verification, and implementation workflow under a single language. In this paper, we present the extensions to Zélus that comprise MARVeLus and demonstrate its functionality by modeling, verifying, and implementing a software component on a scale-model autonomous vehicle.

2 MARVeLus

2.1 Modeling with Synchronous Programming

MARVeLus is based on Zélus, which itself is inspired partially by Lustre [16], a synchronous programming language. Synchronous programming languages, such as Lustre [16], Esterel [15], and Lucid [47], operate on the notion of synchronizing program operations to the discrete steps of a logical clock, which forms the basic timekeeping unit of the program [9]. On each clock cycle, new values are computed concurrently for all the program state variables, and are made available for the next cycle [9]. Thus, it is convenient to think of these variables as streams, along with computations that manipulate their behaviors and thus the values they produce in subsequent cycles [16]. As a result, essentially all Lustre constructs are streams, from literals that are constant streams of their values, to streams of functions that are applied point-wise to their argument streams at each clock cycle [13, 16].

We make a distinction between streams and the values they make available, or "emit", at any given moment, and in the formal semantics of MARVeLus write $e \overset{v}{\hookrightarrow} e'$ to denote the stream e that evolves into e' on the next clock cycle, while simultaneously presenting the immediate value v with which to perform the next computation [16, 19]. Although Lustre permits streams that may not emit values on every clock cycle, we assume that all streams have compatible clocks, an assumption that is statically enforced by Lustre's clock calculus [16].

Lustre programs, and by extension, MARVeLus programs, enjoy some protection from common programming mistakes, both inherently and by static analysis. Streams may only be defined in terms of an initial value and a finite, constant number of previous states. As a result, random access, and thus buffering of an unbounded number of stream values is prohibited, preventing memory leaks. This is desirable for computationally-limited embedded systems, as it makes memory usage deterministic [13, 16]. Synchronous programs also enforce causality, which prevents physically infeasible circular dependencies [21]. This is statically checked by Lustre and its derivatives at compile-time and can catch infinite loops and uninitialized variables [9, 16].

Along with the benefits inherent to synchronous languages, the Zélus language provides us with a complete compilation and simulation toolchain [7], which we extend in MARVeLus with verification and robotics. Source code written in Zélus is first translated into OCaml, which enables our compiler extensions to leverage powerful features such as direct compilation into executable binaries and inclusion of external C functions. The compiler itself is open-source and is also written in OCaml, which makes incorporating new language features a more straightforward process. In addition, its support for modeling hybrid systems, or those that combine both continuous- and discrete-time program evolution [14], gives us the opportunity to eventually expand verification into the continuous space.

Synchronous programming has already demonstrated a history of industry adoption in safety-critical applications. SCADE [20], a tool developed from Lustre, has been used to design avionics on commercial airplanes and high-reliability industrial controls, while Esterel has been used to model the behavior of digital logic for the design of integrated circuits [9, 10]. Thus, the market demand for safer synchronous languages is present and may continue to increase as safety-critical programming becomes more accessible and programs in other fields begin to be held to higher standards of reliability [18].

2.2 Verification with Refinement Types

We enable verification by implementing a refinement type system [42, 46], which allows the user to express desired program properties as annotations on the existing types that program data may already inhabit. Refinement types allow the user to add additional constraints to the type-checking problem already being solved at compile-time by conventionally typed programming languages. These constraints are expressed as boolean predicates that describe properties of the terms inhabiting the type [42]. For instance, the type refinement `{v: int | v >= 0}` describes the type that can only be inhabited by non-negative integers, where `v` is

a variable representing an arbitrary term of the type [32]. In other words, the aforementioned type is inhabited by all integers v such that `v >= 0`. Refinement types are a subset of the overarching dependent type theory, such as that implemented in the F* language [43]. Although both allow type specifications to depend upon program values, one flavor of refinement types—known as liquid types [42, 46]—restrict predicates within type annotations such that they can be decidable using a Satisfiability Modulo Theory (SMT) solver, such as Z3 [22]. This prohibits certain constructs within predicates, such as interpreted functions or quantifiers, but gives the programmer reassurance in the result the type-checker produces [32, 42].

2.3 Execution on Robots

One of our novel additions is the robotics component of MARVeLus, comprising modifications to the Zélus runtime to allow it to share data with external processes managing the real-time operation of robot peripherals. Communications is handled using the Lightweight Communications and Marshalling (LCM) [34] protocol which allows MARVeLus execution to occur wirelessly across the local network, or locally onboard the robot. The former configuration allows more flexibility in switching between physical hardware and simulation, while the latter allows for less network latency. Due to potential performance limitations on embedded platforms, the verification component of MARVeLus is handled on a separate computer. To simulate an autonomous vehicle, we have modified a small wheeled robot with additional software components that mimic realistic vehicle dynamics at a small scale.

3 Motivating Example: Autonomous Vehicle Braking

We examine the use of MARVeLus in verifying a component of Adaptive Cruise Control. Adaptive Cruise Control (ACC) is a driver-assist feature that functions like conventional cruise control, but also monitors the lane ahead for traffic and automatically maintains a safe following distance [38, 39]. ACC may also need to brake in response to stopped vehicles or obstacles on the road in order to prevent a collision [38]. Above all, ACC must avoid collisions whenever possible, and as a result, the absence of collision (expressed as the follower vehicle always remaining behind the leader) is a typical safety property to verify [35].

Formal verification of this safety-critical software component has been explored before, for instance using differential dynamic logic and an automated proof assistant [35]. However, using MARVeLus, we demonstrate automated verification of implementation-level software that is executable on real hardware. We introduce a simplified version of the pre-collision braking component of ACC, modeled after an existing local lane control model [35]. In this system, the

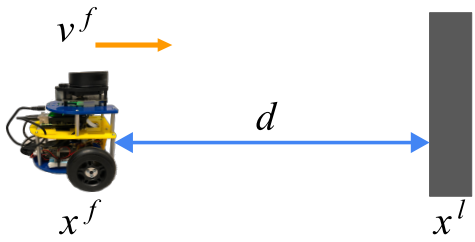

Figure 1. The autonomous braking system visualized. x^l is the position of the obstacle, which our vehicle, at x^f approaches with velocity v^f

software has control of the vehicle's brakes, up to the vehicle's maximum capabilities, and has accurate knowledge of the vehicle's distance to the lead vehicle. Both the lead and ACC vehicle are simplified to be infinitesimal points along a one-dimensional line representing the lane. As with the original model [35], we note that vehicle length can be accounted for by shifting the points to align with the front of the follower and the rear of the lead vehicle or obstacle (Fig. 1).

We model the situation in which the lead vehicle is stationary, or the follower vehicle is approaching an obstacle. Our model (Fig. 2) implements discretized dynamics evolving with a small time step dt and a basic braking algorithm where, upon sensing an obstacle too close to the vehicle, applies full braking force. x^f is the position of the follower vehicle, with initial position x_0^f, v^f is its velocity with initial velocity v_0^f, a^f is the acceleration, which depends on the following constants: braking acceleration b and lead vehicle position x^l. The quantity $x^f + \frac{(v^f)^2}{2b} + v \cdot dt$ denotes the position of the follower's vehicle at which braking must occur to avoid a collision, taking into account braking distance and a reaction time of dt. This is similar to the constraint on follower vehicle behaviors in the original local lane control model, which restricts the acceleration choices of the follower vehicle when it must brake to avoid a collision [35]. The additional term $\frac{v \cdot dt}{2}$, separated for emphasis, represents the over-estimation of distance traveled due to the discretized dynamics of our system, as opposed to the continuous dynamics in the original model.

We adopt the convention that positive acceleration indicates acceleration along the forward direction of vehicle travel; consequently, a becomes negative when the vehicle is decelerating due to braking. The addition of the $\max(0, \cdot)$ condition on velocity ensures that the vehicle's brakes cannot continue accelerating the vehicle in the negative direction once stopped, which models the expected behavior of real brakes. The last line denotes the safety condition, which is that at all times, the follower vehicle's position is behind that of the leader.

$$x^f_{n+1} = x^f_n + v^f_n \cdot dt$$
$$v^f_{n+1} = \max(0, v^f_n + a^f_n \cdot dt)$$
$$a^f_{n+1} = \begin{cases} -b & \text{if } (x^f_n + \frac{(v^f_n)^2}{2b} + v^f_n \cdot dt + \frac{v^f_n \cdot dt}{2} > x^l) \\ 0 & \text{otherwise} \end{cases}$$
$$\forall n \,.\, x^f_n < x^l$$

Figure 2. Discretized autonomous braking system, with a stationary obstacle and safety condition.

Throughout these examples, we make simplifying assumptions on the physical characteristics of the vehicle (such as braking acceleration remaining constant), though we note that these can be accounted for by either constructing these simplifications to be an over-approximation of the original system, or by implementing a higher-fidelity model, which may require more effort.

4 Verification using MARVeLus

4.1 Refinement Types in a Synchronous Language

One of the challenges of developing a refinement type system for a synchronous language is the ever-present balance between having types that are specific enough to reason meaningfully with terms, but also broad enough so that a stream's behavior can be fully described with a single time-invariant type, which is necessary for safety proofs. Thus, refinement types must capture the stream's properties at not only a specific moment, but for all time. Presently, our type system is able to check properties that are globally true for all time, noting that safety properties typically fall under this category.

We designed a set of typing rules based on our characterization of Lustre stream behaviors (Fig. 3), of which we present a selection of those relevant to our examples (Fig. 4). Example stream behaviors include constant streams, basic operators and functions applied point-wise to streams, branching streams that conditionally take values from two other streams, and sequential compositions of streams. We account for variables that can be assigned to streams, and also admit streams that are recursively defined in terms of variables, so long as they adhere to Lustre's existing causality rules [31].

Although all values in Lustre programs are treated as streams, even literals, we denote stream terms with the stream annotation solely in the typing rules for clarity. We also note that the temporal operator $\square$ is used similarly to denote properties that must be true for all values of a given stream. The distinction becomes apparent in the (T-APP) rule which lifts non-stream functions into streams that can be applied to stream arguments. Some MARVeLus constructs already appear in other refinement-typed languages, and some could even be ported directly with minimal modification, such as (T-LETREC) [32]. However, some rules required a more thorough consideration of behaviors unique to their stream counterparts.

Combinatorial functions (which do not carry any state information), such as arithmetic operators, are handled similarly to Lustre, in that they are imported from the non-stream base language and extended as a constant stream of functions that operates point-wise on individual values within streams, rather than as a single function that operates on entire streams [16, 31]. This is shown in the (T-APP) rule by the extension of a typical non-stream function of type $x : \alpha \to \tau$ to a stream of functions f, which for each value within the stream f is applied to the value of the argument stream e from the corresponding time step.

Another useful feature of Lustre is the sequential composition of streams using the fby operator, which is typed with the (T-FBY) rule. The stream e_1 fby e_2 represents a stream consisting of the initial value of e_1, and then the entire stream e_2 delayed by a single cycle. Together with let rec (T-LETREC), one can define recursive streams, as the single cycle delay prevents circular definitions [31]. Thus, the stream let rec $x = x + 1$ in x is disallowed, but can be rewritten as let rec $x = x_0$ fby $x + 1$ in x, which properly initializes the stream with a starting value x_0 and delays the recursive definition by one cycle.

We also allow product types, or n-tuples, which resemble system state vectors and allow multiple values to be consolidated into a single stream. This is vital for specifying properties that relate multiple system state variables and are an extension of dependent pairs used in other dependently-typed languages [32, 43]. Typing of pairs is handled by the (T-PAIR) rule, which places all refinement typing obligations on the last element of the pair. This can then be extended to products with n elements.

A notable departure from conventional refinement typing rules occurs with branching, such as if e_c then e_t else e_f, where the stream may switch between the value of e_t and e_f depending on the value of e_c in a given clock cycle. Programs with branching in conventional languages, such as the expression if c then t else f requires some path sensitivity to verify [46]. This involves additional assumptions to account for the fact that t or f are only reachable when c is true or false, respectively. This prevents the type checker from being unnecessarily strict by ignoring potentially unsafe behavior that occurs in an unreachable branch. For example, if $x < 0$ then $-x$ else x computes the absolute value of an integer x. Proving that the expression only returns non-negative numbers requires knowing that $-x$ is only reachable if $x < 0$; it is impossible to prove the property on $-x$ alone [32]. Extending this method to streams involves making similar assumptions, but must also account for the fact that the conditional, e_c, is a stream as well; thus the resultant stream is an amalgamation of values from both

$e ::=$	
$\mid c$	Constants
$\mid x$	Variables
$\mid$ let $x : \tau = e_1$ in e_2	Local Definitions
$\mid$ let rec $x : \tau = e_1$ in e_2	Recursive Definitions
$\mid e_1\ e_2$	Application
$\mid$ if e_c then e_t else e_f	Branching
$\mid e_1$ fby e_2	Stream Composition
$\mid (e_1,\ e_2, \ldots e_n)$	Products
$\mid$ robot_str $k\ e \mid$ robot_get k	Robot I/O
$\mid e_1$ models e_2	Modeling
$b ::= int \mid float \mid bool$	Base Types
$rv ::= v : b$	Refinement Variables
$\mid (v_1 : b_1) \times \ldots \times (v_n : b_n)$	
$\tau ::= b \mid \{rv \mid \phi\} \mid \{rv \mid \Box\phi\}$	Refinement Types

Figure 3. The Syntax of Streams, where e is a stream expression, k is a string constant used as an identifier, and ϕ are predicates within the quantifier-free logic.

e_t and e_f conditioned on the value of e_c on each cycle. To account for this, we insert assumptions about e_c into the refinement predicates for each of the branches in (T-IF) that ensures the original predicates only need to hold if they are reachable in a given moment based on the value of e_c.

4.2 Type Checking

MARVeLus uses the Z3 SMT solver [22] to determine the validity of verification conditions generated using the aforementioned typing rules. Similarly to Liquid Haskell, each type-checking obligation is considered using a subtyping relation, in which a precise type is synthesized for the term and then checked to be a subtype of the desired type [32]. As a result, type checking becomes a check of validity of the implication statement $\phi_s \Rightarrow \phi_t$ where ϕ_s is the predicate of the synthesized type, and ϕ_t is the predicate of the type to be checked. We also note that streams are constructed via manipulations on streams as a whole (i.e., $x + 1$ refers to the stream where each values is 1 more than the corresponding value of the stream x), properties such as $\Box(x \geq 0)$ are also defined holistically in terms of the stream, and a verification condition may resemble $\forall x\ .\ x \geq 0$. This can be verified via SMT by checking the validity of $x \geq 0$, where x is left free. However, it is often useful to find concrete counterexamples, so we negate the verification condition and check for its unsatisfiability. If the negated formula is found to be unsatisfiable, the original formula is valid. If it is satisfied, the solver is able to provide a model, which informs the user of a variable assignment that leads to a counterexample.

To demonstrate the type-checking process, consider a simple MARVeLus program which defines a pair of integer streams x and y (Fig. 5). The stream x increases with each time step until it reaches 6, at which point it resets back to 0, resulting in a stream that cycles through the integers 0-6, while the stream y is always double the value of x. These streams are defined concurrently through initial values (Line 4) and recursive definitions (Line 5-6). The type specification (Line 3) stipulates that neither stream goes negative, with the safety property v >= 0 && w >= 0. It is clear that this program would satisfy this property at all times. If we were to modify the specification or program so that the property no longer holds, we would encounter a typing error which would halt the compilation process. In addition, we would be presented a counterexample as a set of variable assignments that violates the specification. This aids the developer in either correcting a coding mistake or refining the specification if the counterexample is spurious.

The recursive nature of stream definitions lends itself well to proving invariance of the safety property via induction. This is facilitated somewhat by the let rec and fby constructs. The (T-LETREC) rule allows the type checker to assume that the stream variables satisfy the invariant in a previous cycle (that is, we can assume $x \geq 0$ and $y \geq 0$), providing an induction hypothesis to prove invariance on the recursive definition (that is, it must then show that (if $(x > 5)$ then 0 else $(x+1)) \geq 0$ and $2 \cdot x \geq 0$). Meanwhile, (T-FBY) requires the property to be proven for both the left-hand stream (the initial condition) and the right-hand stream (the recursive definition), thus requiring proof of invariance in both the base case and inductive step.

Verification condition generation proceeds in a syntax-directed fashion based on the typing rules, using the aforementioned subtyping relation as the main tactic for refinement type checking. In our above example, the verification condition to be checked for validity is

$$\begin{aligned}&(((x \geq 0) \wedge (y \geq 0)) \Rightarrow ((0 \geq 0) \wedge (0 \geq 0)))\\ \wedge&(((x \geq 0) \wedge (y \geq 0)) \Rightarrow\\ &\qquad ((\text{ite}(x > 5,\ 0,\ x + 1) \geq 0) \wedge (2 \cdot x \geq 0)))\end{aligned}$$

where the right-hand side expressions of each implication are the left- and right-hand side streams of the fby expression substituting for v and w in the original refinement predicate v >= 0 && w >= 0, and ite is the if-then-else operator in the SMT logic. This is then negated and passed to the SMT solver to check for unsatisfiability and search for possible counterexamples.

5 MARVeLus Implementation of Autonomous Braking

We model the autonomous braking system from Figure 2 in MARVeLus (Fig. 6). Being a deterministic executable language, we must assign concrete values to the free variables

$$\frac{}{\Gamma \vdash c : \{v : b \text{ stream} | \Box(v = c)\}} \text{(T-CONST)} \qquad \frac{\Gamma \vdash e_c : \text{bool stream} \qquad \Gamma \vdash e_t : \{v : b \text{ stream} \mid \Box(e_c \Rightarrow \phi)\} \qquad \Gamma \vdash e_e : \{v : b \text{ stream} \mid \Box(\neg e_c \Rightarrow \phi)\}}{\Gamma \vdash (\text{if } e_c \text{ then } e_t \text{ else } e_e) : \{v : b \text{ stream} \mid \Box\phi\}} \text{(T-IF)}$$

$$\frac{\Gamma \vdash s_1 : \{v : b \text{ stream} \mid \Box\phi\} \qquad \Gamma \vdash s_2 : \{v : b \text{ stream} \mid \Box\phi\}}{\Gamma \vdash s_1 \text{ fby } s_2 : \{v : b \text{ stream} \mid \Box\phi\}} \text{(T-FBY)}$$

$$\frac{\Gamma \vdash f : (x : \{v : b_1 \mid \phi_1\} \to \{w : b_2 \mid \phi_2\}) \text{ stream} \qquad \Gamma \vdash e : \{v : b_1 \text{ stream} \mid \Box\phi_1\}}{\Gamma \vdash f\ e : \{w : b_2 \text{ stream} \mid \Box[e/x]\phi_2\}} \text{(T-APP)}$$

$$\frac{\Gamma \vdash e_1 : b_1 \text{ stream} \qquad \Gamma \vdash e_2 : \{w : b_2 \text{ stream} \mid \Box[e_1/v]\phi\}}{\Gamma \vdash (e_1,\ e_2) : \{(v : b_1 \text{ stream}) \times (w : b_2 \text{ stream}) \mid \Box\phi\}} \text{(T-PAIR)} \qquad \frac{\Gamma, x : \tau_1 \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let rec } x : \tau_1 = e_1 \text{ in } e_2 : \tau_2} \text{(T-LETREC)}$$

Figure 4. Selected Stream Typing Rules

```
1  let node main () =
2    let rec ((x, y):
3      {(v:int)*(w:int) | v>=0 && w>=0}) =
4        ((0, 0) fby
5        ((if (x > 5) then 0 else (x + 1)),
6          2*x))
7      in (x,y)
```

Figure 5. A simple MARVeLus program.

in the system (Lines 1-2). To determine reasonable values, we scaled figures from real automobiles down to our small wheeled robot. For instance, we based our braking acceleration on the 90th percentile braking guidance given by the American Association of Highway Transportation Officials, of $-11.2\,\text{ft/s}^2$ ($-3.4\,\text{m/s}^2$) [26], which scaled down to $-0.136\,\text{m/s}^2$. The starting velocity of 45 miles per hour was likewise scaled to 0.8 m/s. We then determined the obstacle distance such that the vehicle had ample distance to coast for some time before braking, and would avoid a collision if the controller functioned as intended.

Line 4 defines the synchronous node that simulates the vehicle; it has no inputs because the model is self-contained. Line 5 defines the recursive state vector stream with variables `df`, `vf`, and `af` representing the follower vehicle's distance from the obstacle, velocity, and acceleration. Line 3 begins the type specification of the state vector. A small change from our original model was performing the transformation $d = (x^l - x^f)$, where d represents the distance remaining between the vehicle and obstacle. This models the distance sensor used on the real robot.

Lines 6-10 define the type annotation on the state vector, which forms the safety property to be checked at compile time. The first portion of the specification (Line 6) is the main safety property to verify, and requires that the remaining distance must always be positive; a negative or zero distance indicates a collision. However, this alone is insufficient for proving safety, as it is not a property that can be inductively proven—the vehicle may be placed an infinitesimal distance away from the obstacle and still be expected to stop from full speed. This is not a condition in which the system is reasonably expected to be safe, so we add some additional assumptions to the invariant. Other spurious counterexamples may appear, such as the follower vehicle accelerating to physically impossible speeds. Thus, several other invariants are needed to constrain the problem.

The property on Line 7 ($d^f - \frac{(v^f)^2}{2b} - \frac{(v^f)\cdot dt}{2} > 0$) adds the assumption that the vehicle always has time to stop if it were to brake immediately; otherwise the controller would not have been able to prevent a collision in the first place. The specification becomes more precise and we avoid some spurious counterexamples, but it still does not capture the safe behavior of the controller. This is because the controller has a time delay of dt, which means that at any given time step, the braking acceleration was determined from the previous time step's distance and velocity values. As a result, the system must have knowledge of previous control actions, and must be reassured that they did not place the vehicle into an unsafe situation. Thus, we need an additional invariant, seen on Lines 8-9:

$$\left((d^f - \frac{(v^f)^2}{2b} - v^f \cdot dt - \frac{v^f \cdot dt}{2} \leq 0) \wedge (a^f \leq b)\right) \vee$$
$$\left((d^f - \frac{(v^f)^2}{2b} - v^f \cdot dt - \frac{v^f \cdot dt}{2} > 0) \wedge (a^f \leq 0)\right)$$

This invariant requires that, if the follower vehicle is too close to the leader, its braking acceleration must be at least as strong as b, the minimum deceleration required to bring the vehicle to a stop before crossing x^l. Otherwise, the vehicle is free to coast with zero acceleration, or brake at any rate. The second part is also important in prohibiting large positive accelerations that may place the vehicle into a situation in which it cannot brake in time. It can be shown that this invariant ensures that the controller could have only taken safe actions in the previous time step.

Lines 10-19 comprise the bulk of the executable program, and the portion of the program to be verified in this case.

We add a safety factor of 0.5 m in the controller (Line 17) so that the vehicle stops well short of the obstacle. Due to how the invariant is constructed, we must respect the causal ordering of variable updates—the invariant on acceleration (Lines 8-9) is defined in terms of distance and velocity values at the current time step, which means they must be updated before the acceleration is determined. Otherwise, the variables presented to the type refinement become "out of sync" with one another and may not properly verify. This is a consequence of the discretized dynamics and how the invariant was constructed.

Lines 11-12 and again lines 13-14 demonstrate new syntax meant to facilitate implementation. The construct `x models y` denotes that these variables have two definitions: the model `x` which is used for verification and simulation, and the implementation `y`, which is used only when compiling to robot code. This allows real-world values, such as sensor values, to be substituted for `x` during execution, but provides a concrete model when `y` is not available, such as when verifying or simulating. This brings with it the implicit assumption that `x` does in fact model `y`, but ensuring the model is accurate is left as a responsibility of the end user. Whether `x` or `y` appears in the compiled program is determined by the presence of the `-robot` compiler flag.

Several robot-specific keywords appear throughout the program, such as `robot_str` on Line 19 and `robot_get` on Lines 12 and 14. These allow the program to respectively store and retrieve values from named variables shared with external programs, such as sensor and actuator values. Although `robot_str` has no effect in verification or simulation, `robot_get` is prohibited from appearing in verified code except on the right-hand side of an `x models y` structure. This ensures that the variables remain deterministic during verification.

6 Hardware Implementation

Once the system is verified, there comes the task of executing on actual hardware. The execution runtime of MARVeLus is derived from that of Zélus, with some additional functionality to enable programs to communicate with actual hardware. Currently, we have tested MARVeLus execution on small wheeled robots, which we have extended with additional code simulating the dynamics of a scale-model automobile.

6.1 Hardware

The wheeled robot platform we chose for testing, an M-Bot Mini, was originally designed as a teaching tool for a robotics course, to be deployed en masse to students. As a result, it is a well-documented, modular, and inexpensive platform for testing robotics code outside the classroom. The M-Bot Mini features two wheel motors with quadrature encoders arranged in a differential-drive configuration, a rotating LIDAR sensor, and inertial sensors in the form of an internal accelerometer and gyroscope. The main compute module is a BeagleBone Blue, a single-board computer running Debian, which interfaces with motors, rotary encoders, and intertial sensors, and executes MARVeLus code.

6.2 Software

In a typical robotics application, the robot software would be comprised of three components: the user's compiled MARVeLus code, the robot runtime, and the low-level drivers. In our autonomous vehicle example, we further extend the robot runtime with a kinematic model that simulates the acceleration and braking performance of a realistic automobile at the size scale of the robot. For simplicity, we assume all code below the level of user code is safe and that sensor data is reliable. Communications between the robot runtime, kinematic model, and low-level drivers are handled using replicated key-value stores at each communications endpoint, updated via LCM [34] messages. In our example, all software runs onboard the robot to minimize network latency.

6.2.1 User MARVeLus Code. Due to the limited computational resources on the M-Bot Mini, the user's MARVeLus source code is first verified on a desktop computer. If verification passes, the code is compiled into OCaml through the Zélus compiler, which adds the necessary robot interfaces. The resulting OCaml files are then transmitted to the robot, where they are compiled into a native executable alongside the robot runtime.

6.2.2 Robot Runtime. The robot runtime is comprised of a C library that handles networking between the low-level drivers and running MARVeLus program, and the Zélus runtime that handles synchronous program execution and has been modified with hooks to call the C library functions. The C library code size is kept to a minimum, only being used to directly translate OCaml data types into LCM messages and vice versa, and to maintain the key-value store for variable read and write operations.

6.2.3 Kinematic Model. The kinematic model is written in (unverified) Zélus and simulates the discrete dynamics of the vehicle. The model takes as input the commanded acceleration value, which is the only parameter modifiable by user code. This value is then incorporated into the kinematic simulation to obtain vehicle velocity, which is sent to the low-level drivers to control motor speed. It is assumed that, due to the low mass of the robot and high motor torque, motor response time is negligible compared to the simulated dynamics so it is able to reliably attain the velocities requested by the model. The dynamics are implemented as follows:

$$v_{n+1} = v_n + b_n \cdot dt \qquad v_0 = 0.8$$

where b is the braking acceleration input.

```
let vfi:{v: float | v > 0.} = 0.8; let dt:{v: float | v > 0.} = 0.1;
let b:{v: float | v > 0.} = 0.136; let xl:{v: float | v > 0.} = 5.

let node exec () =
  let rec ((df, vf, af):
    {(d:float)*(v:float)*(a:float) | (d > 0.) &&
        (d -. ((v*.v)  /. (2.0*.b)) -. (v*.dt /. 2.) > 0) && (v >= 0.) &&
        ((((d -. ((v*.v) /. (2.0*.b)) -. (v*.dt) -. (v*.dt /. 2.)) > 0) && (a <= 0.)) ||
         (((d -. ((v*.v) /. (2.0*.b)) -. (v*.dt) -. (v*.dt /. 2.)) <= 0) && (a = -. b)))
    }) = (xl, vfi, 0.) fby
      (let v_next = ((if (vf +. (af *. dt) < 0.) then 0. else vf +. (af *. dt))
                      models (robot_get ("vel"))) in
        (let d_next = ((df -. (v_next *. dt))
                        models (robot_get ("dist"))) in
          (d_next, v_next,
           (if (d_next -. ((v_next *. v_next) /. (2.0*.b)) -.
             (v_next *. dt) -. (v_next *. dt /. 2.)) <= 0.5
               then (-.b) else 0.))))
    in (robot_str ("brake", af)); (df, vf, af)
```

Figure 6. Autonomous Braking MARVeLus program

6.2.4 Low-Level Drivers. The low-level drivers directly communicate with the sensors and actuators on the robot. The drivers implement closed-loop motor speed control and odometry, and obstacle sensing using the LIDAR sensor, all of which are currently assumed to be trusted. Some components of the drivers, such as speed control, are implemented at this level purely for convenience, to simplify the MARVeLus code. Other functions translate incoming variable writes into motor commands, interfacing directly with hardware control libraries on the controller board [6]. These functions also generate their own write events based on sensor data, which are propagated to the other software components. For instance, incoming LIDAR data is used to simulate a forward-looking distance sensor on an autonomous vehicle, which produces a single value representing the robot's distance to the nearest obstacle present in the robot's forward-facing field of view. Future work will explore minimizing unverified code in the low-level drivers, instead implementing more features in the verified MARVeLus portion.

6.2.5 Experimental Results. We ran our Autonomous Vehicle Braking code on the M-Bot Mini in a controlled environment where the robot is placed 5.0 m away from a stationary obstacle. The data obtained from one such trial is plotted in Figure 7, superimposed on data taken from a simulated run of the same MARVeLus code. The behaviors shown in position and velocity are largely similar, though the actual robot stops approximately 0.2m ahead of what the simulation predicted, likely due to a combination of slow LIDAR update rates (approximately 5 Hz) and low temporal resolution due to the large discretization period. We also note the slight increase in velocity at the beginning of the trial due to overshoot in the robot's low-level speed controller, which can be mitigated by improved tuning. The model's acceleration exhibits a seemingly unusual "on-off" behavior, which can be explained by the fact that the controller's definition (Fig. 6) essentially mimics the behavior of a bang-bang controller, as it applies brakes only if the vehicle's distance to the obstacle drops below a given threshold, which varies with velocity. As a result, the velocity sometimes drops enough that the braking threshold jumps ahead of the vehicle's current position, causing it to release the brakes momentarily until it once again crosses it.

7 Related Work

7.1 Cyber-Physical Systems Verification

Hybrid automata [1] and differential dynamic logic [41] have been well-studied for the modeling and verification of cyber-physical systems, with approaches ranging from model checking [2] to semi-automated proof assistants [27], the latter of which was applied to verifying an adaptive cruise control model [35]. Although the theory provides a strong foundation for verification, nondeterminism makes it challenging to produce executable code directly from the model. As a result, translation may be needed [4] to generate executables, which although verifiable [11], increases the trusted code base or may present issues in scalability.

CoCoSim [12] is a Simulink toolbox that allows one to place assume-guarantee contracts in a model to be automatically verified by an external solver, such as Zustre [33]. Although the user-facing interface is Simulink, the back-end

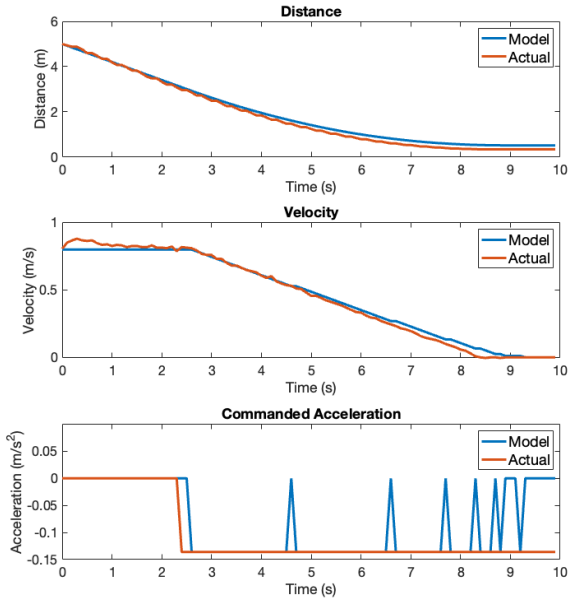

Figure 7. Comparison between simulated and actual values.

compiler first translates Simulink models into Lustre, where verification takes place. As with other presently known Lustre verification methods, this verification takes the form of bounded model checking, which means it may not nessarily produce an exhaustive search of the state space.

The KIND [30] and KIND 2 [17] model checkers provide an SMT-based approach to verifying synchronous programs, by way of k-induction. Invariants can be generated automatically through iterative abstraction refinement, which streamlines the verification process. As with other model-checking approaches, verification may enumerate a large number of, but not necessarily an exhaustive set of possible system states.

There also exist tools for runtime monitoring of CPS, such as RTLola [5] and VeriPhy [11], that allow formal specifications to be compiled into the executable, to be checked for violations at runtime. We note that runtime monitoring and compile-time verification occupy distinct spaces in safe CPS development; while compile-time verifiers like MARVeLus may give developers insight into potential problems before execution, runtime monitors like RTLola may give systems the opportunity to correct for unexpected faults after the fact. Thus, there may be reason to combine both methods to ensure a fallback [11] is available if compile-time assumptions are violated.

7.2 Verifiable and Executable CPS

There exist several projects in this space [3, 11, 23, 29] that allow provably safe CPS code to execute on real systems. This area aims to bridge the gap between abstract, verifiable models and concrete, executable implementation-level code.

VeriPhy [11] is a similar project that provides verified, executable programs for cyber-physical systems. In fact, the VeriPhy executable framework was demonstrated on a wheeled ground robot much like our own. Additionally, VeriPhy provides end-to-end verification with verified compilation to machine code. While the goals of VeriPhy and MARVeLus are similar—verified, executable CPS—the method in which they accomplish this differ. While VeriPhy generates a verified sandbox controller and runtime monitors from abstract specifications as a fallback for user-supplied software, MARVeLus verifies the user-supplied software itself. This way, MARVeLus avoids the need to carry both the user implementation and the synthesized controller. Additionally, it gives the user the opportunity to refine their initially "untrusted" code into a trusted controller with the flexibility of defining verified, customized controller behaviors. Furthermore, direct verification of source code may provide improved traceability of counterexamples encountered.

Drona [23] models high-level behavior as discrete state machines in the P language [24] and system or environment assumptions in Signal Temporal Logic (STL) [37]. In contrast to the types-based verification of MARVeLus, Drona enforces safety of P code using a combination of model checking at compile-time, and monitoring of safety properties in STL at run-time. However, the authors point out that their use of model checking might not provide a complete search of the system's state space, and that the STL specifications used for monitoring are not sound.

CyPhyHouse [29], based on the Koord [28] language, is a robotics stack aimed at verifying multi-agent, possibly heterogeneous, robotics systems. The high-level Koord language allows specification of tasks and associated safety requirements. Additionally, the language features hybrid modeling, with distinct stages of continuous environment and discrete program evolution, in a similar fashion to Zélus. Similarly to MARVeLus, CyPhyHouse uses an SMT-based approach to verify inductive invariants and thus prove safety. The runtime component contains middleware to allow the executing Koord program to interact with platform-specific controllers, such as low-level motion control on a vehicle, and ultimately robot hardware, via ROS. While both MARVeLus and CyPhyHouse enable verification and execution on robotics platforms, the software in each language operates at different levels of the robotics stack, with CyPhyHouse handling high-level planning and coordination tasks, and the current iteration of MARVeLus handling the platform-specific controller code that appears to be unverified in CyPhyHouse.

ROSCoq [3] leverages Coq's dependent type system to specify and verify robotics programs and similarly leverages dependent pairs to capture properties involving relationships

between multiple state variables. In ROSCoq, robots are modeled as distributed message-passing systems, with sensors and actuators acting as nodes in a network, analogous to the way in which real robot components may communicate in the Robot Operating System (ROS) [40].

We also note the work of the Vélus project [13], a formally verified compiler for Lustre, and draw the distinction between a *verified compiler*, such as Vélus, and a *compiler that verifies*, like MARVeLus. The Vélus project is primarily focused on formalizing the process of compiling a synchronous language into an executable imperative one, while our work aims to formalize the semantics of the language in which the user writes.

7.3 Verification through Type Checking

The Curry-Howard Correspondence allows one to essentially turn a type checker into a theorem prover, by encoding propositions as type refinements. Languages such as Liquid Haskell [32] and F* [43] put this principle into practice allowing programmers to include automatically-checkable specifications in their code to be proven at compile-time. As a result, dependently-typed languages have already been applied towards the formal verification of safety-critical software [3, 25, 36]. For instance, Π4 uses type refinements to formally verify properties on executable programs for network equipment [25]. Meanwhile, ROSCoq [3] and VeriDrone [36] use the Coq proof assistant [44], and by extension dependent types, to verify properties in robotics applications. To our knowledge, MARVeLus is the first to combine verification via types with a synchronous programming language.

8 Future Work

A particularly attractive route for improvement would be the implementation of automata and hybrid model simulation, which are hallmarks of the Zélus language [8] and would open up the possibility for vastly more expressive verified programs. The former would allow one to implement finite state machines in controllers, and even emulate certain physical effects such as phase transitions or hysteresis. The latter would make physical models more realistic by introducing continuous dynamics into models and leveraging the methods Zélus uses for reconciling continuous and discrete components in hybrid systems. As a result, the implementation of these components would bring MARVeLus closer to the expressive power of systems already modeled using hybrid automata [1] or differential dynamic logic [41], with the added benefit of direct compilation into executables and code modularity.

End-to-end safety requires not only verification of user code, but also verification of the compiler [13] and runtime monitoring [5]. Verified compilation ensures that the code being verified is indeed the code being executed, while runtime monitoring can detect violations in assumptions made during verification, or can be used to switch between a higher-performing but unverified controller and a verified fallback [11], such as one that may be compiled using MARVeLus. A possible future direction for MARVeLus development may be incorporating these methods into MARVeLus, enabling it to provide end-to-end assurances of safety.

In our present implementation, we still rely upon unverified C-based drivers in the robot firmware to handle speed control, sensing and communications. Currently, MARVeLus already has the capability to model and verify discrete controllers including the robot's speed controllers, though a tighter level of integration with the hardware is nevertheless desirable.

We envision MARVeLus as a robotics stack designed to make formal verification more accessible in robotics applications. To facilitate the adoption of MARVeLus, we plan to eventually add support for other robotics platforms and interfaces such as ROS [40], which has greater potential to bring MARVeLus to a wider user base. Additionally, we intend to open-source MARVeLus to invite community contribution and for transparency.

9 Conclusion

We present MARVeLus as a tool for CPS designers to streamline the process of verification and implementation of real systems. By combining the embedded systems modeling capability of synchronous programming with the compile-time verification of refinement types and real-time communications extensions to the Zélus runtime, MARVeLus features formal verification and execution on real CPS in a single programming language. This is accomplished by adapting a liquid types system, such as that of Liquid Haskell, to accommodate the unique behavior of streams in synchronous programs, and extending a synchronous programming language built upon industry-proven principles with such a type checker and additional drivers to communicate with hardware sensors and actuators. As a result, MARVeLus strives to bridge the gap that has often stood between tools intended primarily for CPS verification and those intended for CPS implementation and execution, providing a unified development workflow upon which robotics applications can be built, with formal verification at its core. We intend to continually improve MARVeLus, allowing it to be usable in an increasingly diverse array of programs and applications, such that we may help to make designing safe CPS ever more accessible.

Acknowledgments

We would like to thank Peter Gaskell and the ROB 550 course staff for providing the M-Bot Mini robots used in our experiments.

References

[1] Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei Hsin Ho. 1993. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, G. Goos, J. Hartmanis, Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel (Eds.). Vol. 736. Springer Berlin Heidelberg, Berlin, Heidelberg, 209–229. https://doi.org/10.1007/3-540-57318-6_30 Series Title: Lecture Notes in Computer Science.

[2] R. Alur, T.A. Henzinger, and Pei-Hsin Ho. 1996. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering* 22, 3 (March 1996), 181–201. https://doi.org/10.1109/32.489079

[3] Abhishek Anand and Ross Knepper. 2015. ROSCoq: Robots Powered by Constructive Reals. In *Interactive Theorem Proving*, Christian Urban and Xingyuan Zhang (Eds.). Vol. 9236. Springer International Publishing, Cham, 34–50. https://doi.org/10.1007/978-3-319-22102-1_3 Series Title: Lecture Notes in Computer Science.

[4] Stanley Bak, Sergiy Bogomolov, and Taylor T. Johnson. 2015. HYST: a source transformation and translation tool for hybrid automaton models. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*. ACM, Seattle Washington, 128–133. https://doi.org/10.1145/2728606.2728630

[5] Jan Baumeister, Bernd Finkbeiner, Sebastian Schirmer, Maximilian Schwenger, and Christoph Torens. 2020. RTLola Cleared for Take-Off: Monitoring Autonomous Aircraft. In *Computer Aided Verification*, Shuvendu K. Lahiri and Chao Wang (Eds.). Vol. 12225. Springer International Publishing, Cham, 28–39. https://doi.org/10.1007/978-3-030-53291-8_3 Series Title: Lecture Notes in Computer Science.

[6] BeagleBoard. [n. d.]. Robot Control. https://github.com/beagleboard/librobotcontrol/releases accessed 09/12/2022.

[7] Albert Benveniste, Timothy Bourke, Benoit Caillaud, Jean-Louis Colaco, Cedric Pasteur, and Marc Pouzet. 2018. Building a Hybrid Systems Modeler on Synchronous Languages Principles. *Proc. IEEE* 106, 9 (Sept. 2018), 1568–1592. https://doi.org/10.1109/JPROC.2018.2858016

[8] Albert Benveniste, Timothy Bourke, Benoît Caillaud, and Marc Pouzet. 2011. A hybrid synchronous language with hierarchical automata: static typing and translation to synchronous code. In *Proceedings of the ninth ACM international conference on Embedded software - EMSOFT '11*. ACM Press, Taipei, Taiwan, 137. https://doi.org/10.1145/2038642.2038664

[9] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. 2003. The synchronous languages 12 years later. *Proc. IEEE* 91, 1 (Jan. 2003), 64–83. https://doi.org/10.1109/JPROC.2002.805826

[10] Gérard Berry. 2008. Synchronous Design and Verification of Critical Embedded Systems Using SCADE and Esterel. In *Formal Methods for Industrial Critical Systems*, Stefan Leue and Pedro Merino (Eds.). Vol. 4916. Springer Berlin Heidelberg, Berlin, Heidelberg, 2–2. https://doi.org/10.1007/978-3-540-79707-4_2 Series Title: Lecture Notes in Computer Science.

[11] Rose Bohrer, Yong Kiam Tan, Stefan Mitsch, Magnus O. Myreen, and André Platzer. 2018. VeriPhy: verified controller executables from verified cyber-physical system models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, Philadelphia PA USA, 617–630. https://doi.org/10.1145/3192366.3192406

[12] Hamza Bourbouh, Pierre-Loïc Garoche, Thomas Loquen, Éric Noulard, and Claire Pagetti. 2020. CoCoSim, a code generation framework for control/command applications An overview of CoCoSim for multi-periodic discrete Simulink models. In *10th European Congress on Embedded Real Time Software and Systems (ERTS 2020)*. Toulouse, France. https://hal.archives-ouvertes.fr/hal-02441334

[13] Timothy Bourke, Lélio Brun, Pierre-Evariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. 2017. A Formally Verified Compiler for Lustre. *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2017), 586–601. https://doi.org/10.1145/3062341.3062358

[14] Timothy Bourke and Marc Pouzet. 2013. Zélus: a synchronous language with ODEs. In *Proceedings of the 16th international conference on Hybrid systems: computation and control - HSCC '13*. ACM Press, Philadelphia, Pennsylvania, USA, 113. https://doi.org/10.1145/2461328.2461348

[15] F. Boussinot and R. de Simone. 1991. The ESTEREL language. *Proc. IEEE* 79, 9 (Sept. 1991), 1293–1304. https://doi.org/10.1109/5.97299

[16] P Caspi, D Pilaud, N Halbwachs, and J A Plaice. 1987. LUSTRE: A declarative language for programming synchronous systems. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '87*. ACM Press, Munich, West Germany, 178–188. https://doi.org/10.1145/41625.41641

[17] Adrien Champion, Alain Mebsout, Christoph Sticksel, and Cesare Tinelli. 2016. The KIND 2 Model Checker. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Vol. 9780. Springer International Publishing, Cham, 510–517. https://doi.org/10.1007/978-3-319-41540-6_29 Series Title: Lecture Notes in Computer Science.

[18] Jean-Louis Colaço. 2020. An overview of Scade, a synchronous language for safety-critical software (keynote). In *Proceedings of the 7th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*. ACM, Virtual USA, 1–1. https://doi.org/10.1145/3427763.3432350

[19] Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. 2006. Mixing signals and modes in synchronous data-flow systems. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software - EMSOFT '06*. ACM Press, Seoul, Korea, 73–82. https://doi.org/10.1145/1176887.1176899

[20] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. 2017. SCADE 6: A formal language for embedded critical software development (invited paper). In *2017 International Symposium on Theoretical Aspects of Software Engineering (TASE)*. 1–11. https://doi.org/10.1109/TASE.2017.8285623

[21] Pascal Cuoq and Marc Pouzet. 2001. Modular Causality in a Synchronous Stream Language. In *Programming Languages and Systems*, Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and David Sands (Eds.). Vol. 2028. Springer Berlin Heidelberg, Berlin, Heidelberg, 237–251. https://doi.org/10.1007/3-540-45309-1_16 Series Title: Lecture Notes in Computer Science.

[22] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, C. R. Ramakrishnan, and Jakob Rehof (Eds.). Vol. 4963. Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24 Series Title: Lecture Notes in Computer Science.

[23] Ankush Desai, Tommaso Dreossi, and Sanjit A. Seshia. 2017. Combining Model Checking and Runtime Verification for Safe Robotics. In *Runtime Verification*, Shuvendu Lahiri and Giles Reger (Eds.). Vol. 10548. Springer International Publishing, Cham, 172–189. https://doi.org/10.1007/978-3-319-67531-2_11 Series Title: Lecture Notes in Computer Science.

[24] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. 2013. P: safe asynchronous event-driven programming. *ACM SIGPLAN Notices* 48, 6 (June 2013), 321–332. https://doi.org/10.1145/2499370.2462184

[25] Matthias Eichholz, Eric Hayden Campbell, Matthias Krebs, Nate Foster, and Mira Mezini. 2022. Dependently-typed data plane programming. *Proceedings of the ACM on Programming Languages* 6, POPL (Jan. 2022), 1–28. https://doi.org/10.1145/3498701

[26] Federal Highway Administration. 2014. Guidelines for the Use of Variable Speed Limit Systems in Wet Weather - Safety. https://safety.

fhwa.dot.gov/speedmgt/ref_mats/fhwasa12022/chap_2.cfm accessed 08/10/2022.

[27] Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völp, and André Platzer. 2015. KeYmaera X: An Axiomatic Tactical Theorem Prover for Hybrid Systems. In *Automated Deduction - CADE-25*, Amy P. Felty and Aart Middeldorp (Eds.). Vol. 9195. Springer International Publishing, Cham, 527–538. https://doi.org/10.1007/978-3-319-21401-6_36 Series Title: Lecture Notes in Computer Science.

[28] Ritwika Ghosh, Chiao Hsieh, Sasa Misailovic, and Sayan Mitra. 2020. Koord: a language for programming and verifying distributed robotics application. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (Nov. 2020), 1–30. https://doi.org/10.1145/3428300

[29] R. Ghosh, J. P. Jansch-Porto, C. Hsieh, A. Gosse, M. Jiang, H. Taylor, P. Du, S. Mitra, and G. Dullerud. 2020. CyPhyHouse: A programming, simulation, and deployment toolchain for heterogeneous distributed coordination. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*. 6654–6660. https://doi.org/10.1109/ICRA40945.2020.9196513 ISSN: 2577-087X.

[30] George Hagen and Cesare Tinelli. 2008. Scaling Up the Formal Verification of Lustre Programs with SMT-Based Techniques. In *2008 Formal Methods in Computer-Aided Design*. 1–9. https://doi.org/10.1109/FMCAD.2008.ECP.19

[31] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. 1991. The synchronous data flow programming language LUSTRE. *Proc. IEEE* 79, 9 (Sept. 1991), 1305–1320. https://doi.org/10.1109/5.97300

[32] Ranjit Jhala and Niki Vazou. 2020. Refinement Types: A Tutorial. *arXiv:2010.07763 [cs]* (Oct. 2020). http://arxiv.org/abs/2010.07763 arXiv: 2010.07763.

[33] Temesghen Kahsai, Hamza Bourbouh, and Daniel Larraz. 2022. Zustre. https://github.com/coco-team/zustre accessed 09/11/2022.

[34] LCM Project. [n. d.]. Lightweight Communications and Marshalling. https://lcm-proj.github.io/index.html accessed 11/22/2021.

[35] Sarah M. Loos, André Platzer, and Ligia Nistor. 2011. Adaptive Cruise Control: Hybrid, Distributed, and Now Formally Verified. In *FM 2011: Formal Methods*, Michael Butler and Wolfram Schulte (Eds.). Vol. 6664. Springer Berlin Heidelberg, Berlin, Heidelberg, 42–56. https://doi.org/10.1007/978-3-642-21437-0_6 Series Title: Lecture Notes in Computer Science.

[36] Gregory Malecha, Daniel Ricketts, Mario M. Alvarez, and Sorin Lerner. 2016. Towards foundational verification of cyber-physical systems. In *2016 Science of Security for Cyber-Physical Systems Workshop (SOSCYPS)*. 1–5. https://doi.org/10.1109/SOSCYPS.2016.7580000

[37] Oded Maler and Dejan Nickovic. 2004. Monitoring Temporal Properties of Continuous Signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, Yassine Lakhnech, and Sergio Yovine (Eds.). Vol. 3253. Springer Berlin Heidelberg, Berlin, Heidelberg, 152–166. https://doi.org/10.1007/978-3-540-30206-3_12 Series Title: Lecture Notes in Computer Science.

[38] Aakar Mehra, Wen-Loong Ma, Forrest Berg, Paulo Tabuada, Jessy W. Grizzle, and Aaron D. Ames. 2015. Adaptive cruise control: Experimental validation of advanced controllers on scale-model cars. In *2015 American Control Conference (ACC)*. IEEE, Chicago, IL, USA, 1411–1418. https://doi.org/10.1109/ACC.2015.7170931

[39] Petter Nilsson, Omar Hussien, Yuxiao Chen, Ayca Balkan, Matthias Rungger, Aaron Ames, Jessy Grizzle, Necmiye Ozay, Huei Peng, and Paulo Tabuada. 2014. Preliminary results on correct-by-construction control software synthesis for adaptive cruise control. In *53rd IEEE Conference on Decision and Control*. 816–823. https://doi.org/10.1109/CDC.2014.7039482 ISSN: 0191-2216.

[40] Open Robotics. [n. d.]. ROS - Robot Operating System. https://www.ros.org accessed 11/22/2021.

[41] André Platzer. 2008. Differential Dynamic Logic for Hybrid Systems. *Journal of Automated Reasoning* 41, 2 (Aug. 2008), 143–189. https://doi.org/10.1007/s10817-008-9103-8

[42] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid types. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation - PLDI '08*. ACM Press, Tucson, AZ, USA, 159. https://doi.org/10.1145/1375581.1375602

[43] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2011. Secure distributed programming with value-dependent types. *ACM SIGPLAN Notices* 46, 9 (Sept. 2011), 266–278. https://doi.org/10.1145/2034574.2034811

[44] The Coq Team. [n. d.]. The Coq Proof Assistant. https://coq.inria.fr/ https://coq.inria.fr/, accessed 09/11/2022.

[45] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. 2014. LiquidHaskell: experience with refinement types in the real world. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell - Haskell '14*. ACM Press, Gothenburg, Sweden, 39–51. https://doi.org/10.1145/2633357.2633366

[46] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming (ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 269–282. https://doi.org/10.1145/2628136.2628161

[47] William W. Wadge and Edward A. Ashcroft. 1985. *Lucid, the Dataflow Programming Language*. Academic Press London. http://prog.vub.ac.be/~tjdhondt/ESL/Lucid_files/Lucid,%20the%20Dataflow%20Programming%20Language.pdf

Received 2022-09-08; accepted 2022-10-10