VeriPhy: Verified Controller Executables from Verified Cyber-Physical System Models

Brandon Bohrer
Carnegie Mellon University
USA

Yong Kiam Tan
Carnegie Mellon University
USA

Stefan Mitsch
Carnegie Mellon University
USA

Magnus O. Myreen
Chalmers University of Technology
Sweden

André Platzer
Carnegie Mellon University
USA

Abstract

We present VeriPhy, a verified pipeline which automatically transforms verified high-level models of safety-critical cyber-physical systems (CPSs) in differential dynamic logic (dL) to verified controller executables. VeriPhy proves that *all* safety results are preserved end-to-end as it bridges abstraction gaps, including: *i)* the gap between mathematical reals in physical models and machine arithmetic in the implementation, *ii)* the gap between real physics and its differential-equation models, and *iii)* the gap between nondeterministic controller models and machine code. VeriPhy reduces CPS safety to the faithfulness of the physical environment, which is checked at runtime by synthesized, verified monitors. We use three provers in this effort: KeYmaera X, HOL4, and Isabelle/HOL. To minimize the trusted base, we cross-verify KeYmaera X in Isabelle/HOL. We evaluate the resulting controller and monitors on commodity robotics hardware.

CCS Concepts • **Computer systems organization → Embedded and cyber-physical systems**; • **Software and its engineering → Formal software verification**;

Keywords cyber-physical systems, hybrid systems, formal verification, verified compilation, verified executables

ACM Reference Format:
Brandon 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 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3192366.3192406

PLDI'18, June 18–22, 2018, Philadelphia, PA, USA

ACM ISBN 978-1-4503-5698-5/18/06…$15.00
https://doi.org/10.1145/3192366.3192406

1 Introduction

Safety-critical *cyber-physical systems* (CPSs) such as self-driving cars, trains, aircraft, and ground robots are increasingly autonomous. Hence, providing strong safety guarantees for their implementation is key.

Safety assurance for CPSs poses a unique challenge: Control software for CPSs is constrained by its physical environment (or *plant*). CPS safety emerges as a joint property of the system itself, its operating environment, and assumptions we make about the environment. For example, a ground robot avoiding collisions requires not only correct control, but that other agents behave reasonably and that the robot's brakes are operational. Assumptions about the environment are just as safety-critical as the control logic.

Differential dynamic logic (dL) enables verifying high-level CPS models featuring real arithmetic, nondeterminism, and differential equations. Real arithmetic is crucial for physical models, and abstracts away low-level details of controller implementation, simplifying mathematical reasoning. Nondeterminism both abstracts away irrelevant details of control algorithms and accounts for environmental uncertainty. Differential equations are crucial to describing physics, e.g., continuous motion. To achieve safety guarantees at the executable level, we must not only achieve safety at the source level, but preserve safety as we soundly implement the abstractions of *a)* real controller arithmetic *b)* nondeterministic control and *c)* environment assumptions.

We introduce VeriPhy, an automated pipeline for generation of verified CPS controller executables from verified models, which, as shown in Fig. 1:

- Begins with a dL model verified safe in KeYmaera X [15] and an untrusted controller implementation.
- Extends the ModelPlex [32] feature of KeYmaera X to generate a verified *sandbox* controller to provably detect model violations in the environment and untrusted controller, engaging a verified fallback controller.
- Generates CakeML [25, 45] source which implements real arithmetic soundly with machine-word intervals and is verified against a model of the environment.
- Generates verified executables with the CakeML compiler. Safety (in compliant environments) is preserved.

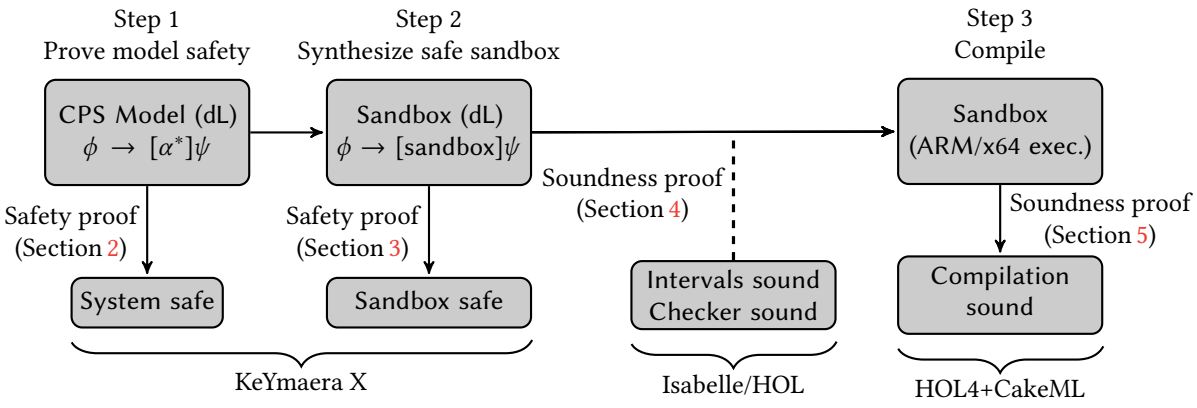

Figure 1. High assurance artifacts and steps in the VeriPhy verification pipeline

To resolve the high-level abstractions of the source model in low-level machine code, VeriPhy exploits multiple theorem provers: KeYmaera X is used to verify hybrid systems models of CPS while HOL4 provides executable-level guarantees via CakeML. The foundations of KeYmaera X and higher-order logic differ greatly; we bridge them to improve trust. In Step 3, the semantic gap from dL's real arithmetic to machine-word interval arithmetic must also be bridged.

Supporting Pipeline Stages. A foundational bridge requires formalizing both hybrid systems and machine-word arithmetic. However, KeYmaera X knows only hybrid systems and HOL4 lacks a hybrid systems formalization. We build the bridge in Isabelle/HOL because it has a dL formalization [2], can express word arithmetic, and has a similar logical foundation to HOL4:

- We first extend KeYmaera X with a proof term exporter.
- We verify a dL proof term checker in Isabelle/HOL.
- We rewrite the proof checker definitions in HOL4 for verified CakeML extraction [35] and, after adding a trusted parser, compilation to a verified executable.

We prove that the checker only accepts proofs of true dL formulas and that interval arithmetic semantics are sound with respect to standard dL semantics, thus bridging dL with higher-order logic and interval arithmetic. We exploit each prover for its strengths: KeYmaera X for its hybrid systems automation, HOL4 for its verified CakeML compiler, and Isabelle/HOL for its dL formalization and word libraries.

Trusted Computing Base. The cost of using multiple provers is increased trusted computing base. We summarize the trusted base and why each component is either simple or can be addressed in future work. We assume:

- The cores of HOL4 and Isabelle/HOL (≈12K lines, well-tested) are sound. Translations of definitions from Isabelle/HOL to HOL4 must be accurate. Some are nontrivial, e.g. we simulate Isabelle/HOL's typeclasses manually in HOL4. Trust in Isabelle/HOL is partial, e.g., a system safety proof is valid unless *both* the KeYmaera X and Isabelle/HOL cores fail at once.
- The hand-written proof term parser is correct. The parser is simple and the checker prints the proof conclusion, which can be inspected as a cross-check.
- Arithmetic solving is sound. Proof-producing solvers [42] fit our proof term framework, but were not used because making them scalable is an open problem.
- Untrusted controllers obey memory isolation with respect to the sandbox controller. Verified isolation is an established but active area of study [34].

The above list shows the components that must be trusted. In return we gain the first automatic pipeline from verified CPS models to verified controller executables.

2 Hybrid Programs

The mathematical essence of a CPS is a *hybrid system* [20] combining discrete and continuous dynamics for control and physics. We review the *hybrid program* (HP) language for hybrid systems and *differential dynamic logic* (dL) [37, 38, 40] for verifying HPs. Table 1 gives their syntax and meaning.

Table 1. Hybrid Programs (HPs)

Statement	Meaning
$\alpha;\beta$	Sequentially composes β after α
$\alpha \cup \beta$	Executes either α or β, nondeterministically
α^*	Repeats α zero or more times
$x := \theta$	Assigns value of term θ to x
$x := *$	Assigns an arbitrary real value to x
$x' = \theta \,\&\, Q$	Continuous evolution[1]
$?Q$	Aborts run if formula Q is not true

Atomic hybrid programs (HPs) comprise deterministic ($x := \theta$) and nondeterministic ($x := *$) assignments, tests ($?Q$),

[1] A continuous evolution along the differential equation system $x' = \theta$ for an arbitrary real duration within the region described by formula Q.

and differential equations constrained to evolution domains ($x' = \theta \,\&\, Q$). Here, θ is a real arithmetic term, possibly mentioning x, and Q a first-order real arithmetic formula. The program connectives provide sequencing ($\alpha;\beta$), nondeterministic choice ($\alpha \cup \beta$), and nondeterministic repetition (α^*). Familiar program constructs like if-then-else are derived:

$$\texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta \overset{\text{def}}{\equiv} ?Q;\alpha \cup ?\neg Q;\beta \tag{1}$$

The formulas of dL are generated by the following grammar ($\sim$ is a comparison operator, $<, \leq, =, \neq, \geq, >$, and θ_1, θ_2 are arithmetic expressions in $+, -, \cdot$ over the reals):

$$\phi ::= \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \phi \rightarrow \psi$$
$$\mid \theta_1 \sim \theta_2 \mid \forall x\, \phi \mid \exists x\, \phi \mid [\alpha]\phi \mid \langle\alpha\rangle\phi$$

The standard logical connectives are as in classical first-order logic (see semantics below). Modal formulas $[\alpha]\phi$ and $\langle\alpha\rangle\phi$ hold if *all* or *some* runs of the (possibly) nondeterministic hybrid program α end in states satisfying ϕ, respectively. We will often use partial correctness assertions $\phi \rightarrow [\alpha]\psi$ stating that when precondition ϕ holds initially, all states reached by running α satisfy the postcondition ψ.

Formal Semantics. The semantics of dL [37, 38, 40] is a Kripke semantics in which the states of the Kripke model are the states of the hybrid system. Let $\mathbb{R}$ denote the set of real numbers and $\mathcal{V}$ denote the set of variables. A state is a map $\omega : \mathcal{V} \rightarrow \mathbb{R}$ assigning a real value $\omega(x)$ to each variable $x \in \mathcal{V}$. The set of all states is denoted by $\mathcal{S}$. We write $\omega \models \phi$ if formula ϕ is true at state ω (Def. 2). The real value of term θ at state ω is denoted $\omega[\![\theta]\!]$.

The semantics of HP α is expressed as a transition relation between states (Def. 1). A differential equation $x' = \theta \,\&\, Q$ can transition between any pair of states connected by a continuous flow φ that respects the differential equations and evolution domain. We write $\varphi \models x' = \theta \,\&\, Q$ to mean that φ is a flow of the differential equation $x' = \theta$ contained within the region Q, see [37, 38, 40] for full details.

Definition 1 (Transition semantics of hybrid programs)**.** The transition relation $[\![\alpha]\!]$ specifies which states ν are reachable from a state ω by operations of α. It is defined as follows:

1. $(\omega,\nu) \in [\![x := \theta]\!]$ iff $\nu(x) = \omega[\![\theta]\!]$, and for all other-variables $z \neq x$, $\nu(z) = \omega(z)$
2. $(\omega,\nu) \in [\![x := *]\!]$ iff $\nu(z) = \omega(z)$ for all variables $z \neq x$
3. $(\omega,\nu) \in [\![?Q]\!]$ iff $\omega = \nu$ and $\omega \models Q$
4. $(\omega,\nu) \in [\![x' = \theta \,\&\, Q]\!]$ iff exists solution $\varphi{:}[0,r] \rightarrow \mathcal{S}$ for $r \geq 0$ with $\varphi(0) = \omega$, $\varphi(r) = \nu$, and $\varphi \models x' = \theta \,\&\, Q$
5. $[\![\alpha \cup \beta]\!] = [\![\alpha]\!] \cup [\![\beta]\!]$
6. $[\![\alpha;\beta]\!] = \{(\omega,\nu) : (\omega,\mu) \in [\![\alpha]\!], (\mu,\nu) \in [\![\beta]\!], \text{exists } \mu\}$
7. $[\![\alpha^*]\!] = [\![\alpha]\!]^*$, the transitive, reflexive closure of $[\![\alpha]\!]$

Definition 2 (Interpretation of dL formulas)**.** Truth of dL formula ϕ in state ω, written $\omega \models \phi$, is defined as follows:

1. $\omega \models \theta_1 \sim \theta_2$ iff $\omega[\![\theta_1]\!] \sim \omega[\![\theta_2]\!]$ for $\sim \in \{=, \leq, <, \geq, >\}$
2. $\omega \models \phi \wedge \psi$ iff $\omega \models \phi$ and $\omega \models \psi$, so on for $\neg, \vee, \rightarrow, \leftrightarrow$
3. $\omega \models \forall x\, \phi$ iff $\nu \models \phi$ for all states ν that agree with ω except for the value (in $\mathbb{R}$) of x
4. $\omega \models \exists x\, \phi$ iff $\nu \models \phi$ for some state ν that agrees with ω except for the value (in $\mathbb{R}$) of x
5. $\omega \models [\alpha]\phi$ iff $\nu \models \phi$ for all ν with $(\omega,\nu) \in [\![\alpha]\!]$
6. $\omega \models \langle\alpha\rangle\phi$ iff $\nu \models \phi$ for some ν with $(\omega,\nu) \in [\![\alpha]\!]$

We denote *validity* as $\vDash \phi$, i.e., $\omega \models \phi$ for all states ω.

Running Example. We introduce an abstract dL model of a ground robot in a corridor, which we reduce throughout the paper to a verified implementation running on commodity robot hardware. The robot must avoid hitting walls and other obstacles but can drive freely otherwise. We model the robot with instantaneous control over its velocity v. This abstraction is faithful as shown in Section 6 because our robot drives slowly relative to its braking power. The controller is time-triggered, i.e., the system delay between controller runs is bounded by some ε.

Formula (2) expresses safety of the model in a dL formula $\phi \rightarrow [(\text{ctrl};\text{plant})^*]\psi$. It says all states satisfying assumptions ϕ lead to safe states (ψ) no matter how long the system loop $(\text{ctrl};\text{plant})^*$ repeats. The program ctrl is a discrete time-triggered controller, while the program plant describes physical environment assumptions as a differential equation.

$$\overbrace{d \geq 0 \wedge V \geq 0 \wedge \varepsilon \geq 0}^{\phi} \rightarrow [(\text{ctrl};\text{plant})^*] \overbrace{d \geq 0}^{\psi} \tag{2}$$
$$\text{ctrl} \equiv (\text{drive} \cup \text{stop});\ t := 0 \tag{3}$$
$$\text{drive} \equiv ?d \geq \varepsilon V;\ v := *;\ ?0 \leq v \leq V \tag{4}$$
$$\text{stop} \equiv v := 0 \tag{5}$$
$$\text{plant} \equiv \{d' = -v,\ t' = 1 \,\&\, t \leq \varepsilon\} \tag{6}$$

Initially, the robot is driving at a safe distance $d \geq 0$ from the obstacles. We also know the system delay $\varepsilon \geq 0$ and maximum driving speed $V \geq 0$. Our safety condition $d \geq 0$ says the robot does not drive through the obstacles. Its controller (3) can either drive or stop (drive $\cup$ stop), followed by setting a timer $t := 0$ which, by (6), wakes the robot controller again after at most time ε. When the test in (4) passes, it is safe to keep driving for ε time, and the robot can choose any velocity $v := *$ up to at most the maximum velocity ($?0 \leq v \leq V$). In each case, the controller is allowed to stop the robot (5) by setting velocity v to 0. Finally, the plant (6) changes the distance according to the chosen velocity v via the differential equation $d' = -v$. Time advances at the rate $t' = 1$, for any duration $t \leq \varepsilon$. The program ctrl; plant can then repeat and the controller can make its next decision. The validity proof of formula (2) is elaborated next.

Proving Safety. The VeriPhy pipeline starts with a *safety proof* in dL of the partial correctness assertion

$$\phi \rightarrow [(\text{ctrl};\text{plant})^*]\psi \tag{7}$$

in the hybrid systems theorem prover KeYmaera X [15].

The proof of formula (7) is input as a proof script for dL in the Bellerophon language [14]. Bellerophon scripts combine high-level automated search procedures from the standard library with manual uses of dL axioms [37, 38, 40]. Typical scripts focus on key system insights, such as invariants for loops and differential equations, and manually assisting automation with challenging sub-problems, like proving statements about real arithmetic. For models like those in this paper, proving is typically automatic once invariants are provided. Interactive proofs from the web-based UI can also be exported to Bellerophon [31], then passed to VeriPhy. **Step 1** of VeriPhy checks the Bellerophon script to establish that the source model has been verified:

Definition 3 (Verified input)**.** The HP $\alpha \equiv (\mathsf{ctrl};\mathsf{plant})^*$ for time-triggered ctrl is *verified* (with φ) if a dL formula $\phi \to [\alpha]\psi$ has been proven valid via a loop invariant φ, i.e., $\phi \to \varphi$, $\varphi \to \psi$ and $\varphi \to [\alpha]\varphi$ have been proven valid.

3 ModelPlex Sandbox Synthesis

To enable abstraction in controller models, dL provides features which make it ill-suited for direct execution, such as nondeterminism. Nondeterministic controller models are a natural fit, however, for *sandboxing* the results of an external unverified controller by monitoring it for compliance with the dL model and executing a safe *deterministic* fallback upon compliance violation. **Step 2** synthesizes from the system safety proof and loop invariant φ such a *sandbox controller* enforcing runtime safety by sandboxing untrusted controllers. Correct-by-construction monitors detect controller bugs and environment model violations [32, Thms. 1+2], invoking verified fallback control or signaling an error, respectively.

The shape of the synthesized sandbox controller is shown in Fig. 2. For clarity, we denote by $\vec{x}$ the vector of all variables in the current program state before executing ctrl; plant, and denote by $\vec{x}^+$ the tentative next state. In all, the sandbox controller performs the following tasks: It *i)* nondeterministically assigns ($\vec{x} := *$) arbitrary values to configuration parameters and initial system state from external sensors in (8), checking that they satisfy the precondition ϕ; *ii)* checks that the untrusted controller decision $\vec{x}^+$ (9) satisfies the monitor formula $\mathsf{ctrlMon}(\vec{x},\vec{x}^+)$ in (10); *iii)* otherwise allows only a safe fallback action (11); *iv)* actuates the decision $\vec{x}^+$ by assigning it to state $\vec{x}$ (12); *v)* models sensing with nondeterministic assignments $\vec{x}^+ := *$ and monitors whether the sensor values comply with the environment model in (14) before storing them for the next iteration with $\vec{x} := \vec{x}^+$ (15).

Lines (10)–(11) correspond to a nondeterministic if-then-else (1) where the else branch in (11) is always allowed. This flexibility becomes important in Section 4 when machine arithmetic introduces uncertainty in the test of (10).

We first discuss the key ingredients of sandboxing: the control monitor ctrlMon (10) for detecting errors in untrusted controllers and the plant monitor plantMon (14) for detecting unexpected environment behavior. We then discuss their incorporation into the verified sandbox controller (Fig. 2) with safe fallback control (11).

$$
\begin{array}{lll}
\mathsf{sandbox} \equiv & & \\
\quad \vec{x} := *;\ ?\phi; & \textit{read initial state} & (8)\\
\quad (& & \\
\quad\quad \vec{x}^+ := \mathsf{extCtrl}(\vec{x}); & \textit{run external control} & (9)\\
\quad\quad \big(?\mathsf{ctrlMon}(\vec{x},\vec{x}^+) & \textit{check if safe action} & (10)\\
\quad\quad\quad \cup\ \vec{x}^+ := \mathsf{fallback}(\vec{x}) \big); & \textit{or fallback control} & (11)\\
\quad\quad \vec{x} := \vec{x}^+; & \textit{actuate action} & (12)\\
\quad\quad \vec{x}^+ := *; & \textit{sense next state} & (13)\\
\quad\quad ?\mathsf{plantMon}(\vec{x},\vec{x}^+); & \textit{check safe env.} & (14)\\
\quad\quad \vec{x} := \vec{x}^+ & \textit{store sensors} & (15)\\
\quad)^* & \textit{repeat} & (16)
\end{array}
$$

Figure 2. Sandbox controller overview

3.1 Controller Monitor Formula

We use nondeterminism in dL controller models to abstract away control algorithm details that are not safety-relevant (e.g., optimizations to save power or ensure smooth travel). Any such details are supplied by the untrusted controller, which can be implemented freely, even in languages that were not designed for verification. The untrusted controller is only known to be safe, however, if it behaves consistently with the verified controller model. ModelPlex [32] synthesizes a real arithmetic formula $\mathsf{ctrlMon}(\vec{x},\vec{x}^+)$ over the model's state variables to check control decisions for compliance with the model. The condition $\mathsf{ctrlMon}(\vec{x},\vec{x}^+)$ is efficiently checked at runtime for concrete values $\vec{x}$ of a start state (e.g., distance sensed before the controller runs) and $\vec{x}^+$ of an end state (e.g., new speed, chosen by the controller).

We give a brief overview of monitor synthesis here, and refer the reader to the literature [32] for full details on how monitor formulas can be automatically synthesized from an input model and verified. ModelPlex composes the safety theorem $\phi \to [(\mathsf{ctrl};\mathsf{plant})^*]\psi$ with offline transformation proofs [32, Lem 4–8], reducing system safety to online monitor compliance. ModelPlex monitors the precondition ϕ when the system starts in state ω_0 (check $\omega_0 \models \phi$ in equation (8) of the sandbox) and the controller monitor condition $\mathsf{ctrlMon}(\vec{x},\vec{x}^+)$ at every observed transition (ω,ν) (check $\mathsf{ctrlMon}(\vec{x},\vec{x}^+)$ in equation (10)). As a result, we get online safety $\nu \models \varphi$ up through the current state ν by [32, Thm 2].

Definition 4 (Compliance)**.** We say transition (ω,ν) *complies* with $\mathsf{ctrlMon}(\vec{x},\vec{x}^+)$, written $(\omega,\nu) \models \mathsf{ctrlMon}(\vec{x},\vec{x}^+)$,

iff $\mathsf{ctrlMon}(\vec{x},\vec{x}^+)$ holds using the values of state ω for plain variables x and the values of ν for variables x^+ in $\vec{x}^+$.

The equations in Fig. 3 illustrate the offline transformation proof for synthesizing controller monitor conditions ctrlMon to check controller implementation correctness. The proof starts at the semantic statement $(\omega,\nu) \in [\![(\mathsf{ctrl};\mathsf{plant})^*]\!]$ and obtains an arithmetic monitor condition $\mathsf{ctrlMon}(\vec{x},\vec{x}^+)$. Condition $\mathsf{ctrlMon}(\vec{x},\vec{x}^+)$ also checks that the plant evolution domain constraint Q holds so that the controller does not itself cause a plant violation upon actuating the output $\vec{x}^+$.

$$
\begin{array}{ll}
(\omega,\nu) \in [\![(\mathsf{ctrl};\mathsf{plant})^*]\!] & \text{Semantical condition} \\
\quad \Updownarrow & \text{by [32, Lem 4]} \\
(\omega,\nu) \models \langle(\mathsf{ctrl};\mathsf{plant})^*\rangle(\vec{x}=\vec{x}^+) & \text{Logical criterion} \\
\quad \Uparrow & \text{by [32, Lem 5]} \\
(\omega,\nu) \models \langle\mathsf{ctrl};\mathsf{plant}\rangle(\vec{x}=\vec{x}^+) & \text{thus } \nu \models \psi \\
\quad \Uparrow & \text{by [32, Lem 6]} \\
(\omega,\nu) \models \langle\mathsf{ctrl}\rangle(\vec{x}=\vec{x}^+ \land Q) & \\
\quad \Uparrow \quad \text{by ModelPlex-generated dL proof, Lemma 1} & \\
(\omega,\nu) \models \mathsf{ctrlMon}(\vec{x},\vec{x}^+) & \text{by online monitoring}
\end{array}
$$

Figure 3. ModelPlex controller monitor synthesis [32]

Monitor Correctness Proof. ModelPlex's synthesized controller monitor conditions are correct by construction [32] from the process in Fig. 3, which guarantees Lemma 1. The controller monitor synthesis process of Fig. 3 starts by obtaining logical criterion $\langle(\mathsf{ctrl};\mathsf{plant})^*\rangle(\vec{x}=\vec{x}^+)$ from the proved property $\phi \rightarrow [(\mathsf{ctrl};\mathsf{plant})^*]\psi$. We denote by $\vec{x}=\vec{x}^+$ component-wise equality between vectors $\vec{x}$ and $\vec{x}^+$.

Lemma 1 (Controller monitor correctness)**.** *The controller monitor* $\mathsf{ctrlMon}(\vec{x},\vec{x}^+)$ *relating control input* $\vec{x}$ *to control output* $\vec{x}^+$ *guarantees that control output* $\vec{x}^+$ *is permitted by the verified control model* ctrl *on input* $\vec{x}$ *and respects the* plant *evolution domain constraint* Q*, i.e.:*

$$\vDash \mathsf{ctrlMon}(\vec{x},\vec{x}^+) \rightarrow \langle\mathsf{ctrl}\rangle(\vec{x}=\vec{x}^+ \land Q)$$

Lemma 1 is a crucial lemma in the sandbox safety proof.

Example. In our running example, the monitor checks the bound variables d, v, t:

$$\langle\mathsf{ctrl}\rangle(\vec{x}=\vec{x}^+ \land Q) \equiv \Big\langle(?d \geq \varepsilon V;\ v := *;\ ?0\leq v\leq V \ \cup\ v := 0);\ t := 0\Big\rangle(d^+{=}d \land v^+{=}v \land t^+{=}t \land t \leq \varepsilon)$$

The offline monitor transformation proofs are implemented as automation in KeYmaera X, outside the trusted core. On the above formula, this monitor formula is output:

$$\mathsf{ctrlMon} \equiv \Big((d \geq \varepsilon V \land 0 \leq v^+ \leq V) \ \lor\ v^+ = 0\Big) \land 0 \leq \varepsilon \land V \geq 0 \land t^+ = 0 \land d^+ = d$$

The monitor checks both possible paths through the controller: the first disjunct captures the test conditions for driving with a new velocity v^+ (nondeterministic assignment $v := *$ followed by test $?0 \leq v \leq V$), whereas the second disjunct captures the emergency stop ($v := 0$), so $v^+ = 0$. The conditions further state that the constants are chosen according to the model assumptions ($0 \leq \varepsilon \land V \geq 0$), that both paths reset their clocks $t^+ = 0$ to correctly measure the duration until the next controller run, and that neither controller path alters the distance measurement, so $d^+ = d$.

3.2 Plant Monitor Formula

ModelPlex also synthesizes a formula $\mathsf{plantMon}(\vec{x},\vec{x}^+)$ which holds only if the values $\vec{x}$ and $\vec{x}^+$ sensed in successive states comply with the plant model. For example, the plant monitor for our ground robot tests that sensed motion is consistent with the maximum speed V.

Exact compliance is typically too restrictive: A differential equation specifies a single exact trajectory for each point, from which realistic sensors will deviate slightly. However, safety proofs need not employ the exact trajectory, but rather often employ *invariant* arguments which specify a broader safety region. In our example, safety eschews the exact trajectory $d^+ = d - vt^+$ in favor of the looser invariant $d^+ \geq v(\varepsilon - t^+)$. It suffices for safety to construct plantMon from the plant model's evolution domain Q (e.g., $t \leq \varepsilon$) and the ODE invariants in the safety proof of Step 1 (e.g., $d \geq v(\varepsilon - t)$). In the sandbox controller Fig. 2, the condition $\mathsf{plantMon}(\vec{x},\vec{x}^+)$ checks that the observed evolution from the sensed values $\vec{x}$ of the previous iteration to the new values $\vec{x}^+$ is within this relaxed safety region. If a plant monitor fails, a violation raises an alarm, upon which best-effort fallback control is typically done. Unlike in the ctrl monitor case, however, fallback controller safety cannot be guaranteed when all of the physical assumptions are violated.

Lemma 2 (Plant monitor correctness)**.** *Let* $(\mathsf{ctrl};\mathsf{plant})^*$ *be verified with* φ*, and let* $\mathsf{ctrlMon}(\vec{x},\vec{x}^+)$ *be a correct controller monitor according to Lemma 1. Then, loop invariant* φ *is preserved when the* plant *monitor* $\mathsf{plantMon}(\vec{x},\vec{x}^+)$ *is satisfied.*

$$\vDash \mathsf{ctrlMon}(\vec{x},\vec{x}^+) \rightarrow [?\mathsf{plantMon}(\vec{x},\vec{x}^+)]\varphi$$

3.3 Fallback Control

Unsafe control choices are detected by the controller monitor and replaced with provably safe fallback control choices. *Any* controller that satisfies the controller monitor can be used for safe fallback according to Lemma 3. Concretely, we take the verified fallback from the controller ctrl, e.g., $v := 0; t := 0$ for our ground robot.

Lemma 3 (Fallback correctness). *Let program* (ctrl; plant)* *be verified with* φ *and let* ctrlMon$(\vec{x},\vec{x}^+)$ *be a controller monitor per Lemma 1. A fallback controller is correct if all runs starting in states satisfying the loop invariant* φ *comply with the controller monitor* ctrlMon$(\vec{x},\vec{x}^+)$*:*

$$\vDash \varphi \rightarrow [\vec{x}^+ := \text{fallback}(\vec{x})]\text{ctrlMon}(\vec{x},\vec{x}^+)$$

3.4 Provably Safe Sandboxing

Theorem 4 says safety results transfer: from the theorem $\phi \rightarrow [(\text{ctrl};\text{plant})^*]\psi$ for program (ctrl; plant)*, we obtain safety of the synthesized sandbox.

Theorem 4 (Sandbox safety). *Let program* (ctrl; plant)* *be verified. Assume a correct controller monitor, correct* plant *monitor, and correct fallback. Then all runs of the* sandbox *program (from Fig. 2) starting in* ϕ *(from Def. 3) are safe (*ψ*):*

$$\vDash \phi \rightarrow [\text{sandbox}]\psi$$

Proof. By dL proof from Lemma 1, Lemma 2, and Lemma 3. □

Running Example. The provable dL formula in Fig. 4 illustrates the controller and plant monitor conditions of our running example embedded into their sandbox.

see (2): $d \geq 0 \wedge V \geq 0 \wedge \varepsilon \geq 0$

$$\begin{array}{lll}
\phi \rightarrow \Big[V := *;\ \varepsilon := *;\ d := *;\ t := *; & // \ \vec{x} := * \\
\quad ?d \geq 0 \wedge V \geq 0 \wedge \varepsilon \geq 0; & // \ ?\phi \\
\quad \Big(\ t^+ := *;\ v^+ := *;\ d^+ := d; & // \ \vec{x}^+ := \text{extCtrl} \\
\quad\quad (\ \ ?\text{ctrlMon}(d,t,v,d^+,t^+,v^+) & \\
\quad\quad\quad \cup\ t^+ := 0;\ v^+ := 0\ ); & // \ \vec{x}^+ := \text{fallback} \\
\quad\quad t := t^+;\ v := v^+; & // \ \vec{x} := \vec{x}^+ \\
\quad\quad d^+ := *;\ t^+ := *; & // \ \vec{x}^+ := * \\
\quad\quad ?\big(0 \leq t^+ \leq \varepsilon \wedge d^+ \geq v(\varepsilon - t^+)\big); & // \ ?\text{plantMon}(\vec{x},\vec{x}^+) \\
\quad\quad d := d^+;\ t := t^+ & // \ \vec{x} := \vec{x}^+ \\
\quad \Big)^* \Big] \psi &
\end{array}$$

see (2): $d \geq 0$

Figure 4. Sandbox of a velocity-controlled ground robot

Truth of the monitor formula implies runtime safety of the CPS, but the monitor formulas and sandboxes are hard to execute until we concretely implement the arithmetic and nondeterministic approximations contained therein.

4 Interval Word Arithmetic Translation

Having shown safety of a sandbox controller in dL, we turn our attention toward correct compilation, the first step of which is to formally justify implementing real numbers with interval arithmetic over machine words. We formalize both real arithmetic and interval arithmetic semantics for dL (in the style of Section 2) and show a soundness theorem: any formula which holds in the interval semantics holds in the real-number semantics. This is a theorem about the semantics of dL, making it outside KeYmaera X's purview. We now transition to Isabelle/HOL to reason about dL semantically.

For a provably sound transition, we develop a bridge from dL proofs in KeYmaera X to semantic truth in Isabelle/HOL, removing KeYmaera X from the trusted base in the process, then verify interval arithmetic soundness within Isabelle/HOL. The bridge builds on an existing formalization of semantics, proof calculus, and soundness of dL [2].

4.1 Proof Export and Import

While using multiple provers in VeriPhy simplifies reasoning across domains (from hybrid systems to machine code), it places additional correctness demands: *i)* shared definitions must have the same semantics in all provers and *ii)* all provers must be sound. We address both problems simultaneously by cross-verifying KeYmaera X into Isabelle/HOL. We implemented proof-term export for KeYmaera X and a verified proof-term checker in Isabelle/HOL.

The proof checker uses a deep embedding of dL [40] in Isabelle/HOL. We extend a previous [2] soundness proof for the core dL calculus theory [40] with more mature formalizations of features needed for practical proofs, such as *nondeterministic* assignments, *systems* of differential equations, variable renaming, and sequent calculus. These extensions are tied together by a proof-checking function formalized in Isabelle/HOL which takes a KeYmaera X proof term and returns the theorem proved by it, if any:

```
pteval : pt → rule option
```

The result is a KeYmaera X proof state, i.e. an arbitrary derived `rule`. Proven dL formulas are rules with no premisses.

Theorem 5 (Proof-checker soundness). *If all real arithmetic subgoals are valid and the proof checker accepts the proof term, the output derived rule is sound, i.e.:*

```
arith_valid pt ∧ pteval pt = Some rule → sound(rule)
```

This assumes all arithmetic goals in the proof are true. They are decidable, but expensive to check in practice. Verified real arithmetic proving is an active research area in its own right [19, 33, 42], which we leave as future work.

KeYmaera X is actively developed, so our goal is not to support all proofs. Our checker supports all sandbox safety proofs in this paper, on the scale of ≈200,000 proof steps.

Tests with shorter proofs indicate these would take days to check in Isabelle/HOL. We generate an executable checker instead, currently hand-translated to HOL4 for extraction because its verified CakeML extractor [35] is mature, while Isabelle/HOL's [21] cannot yet target machine code. Combined with a trusted parser, it rechecks KeYmaera X proofs

quickly (e.g., 6.5s for our example). In all, our extensions to the dL soundness proof are ≈7,000 lines of Isabelle/HOL proof based on 15,000 for the initial proof.

4.2 Interval Arithmetic Translation

We have soundly transitioned from proofs of system safety in KeYmaera X to the truth of system safety according to the semantics of dL in Isabelle/HOL. The standard semantics of dL feature arithmetic on real numbers, which are crucial for physics but ill-suited to efficient execution. Next, we soundly approximate the real semantics with a computable 32-bit integer interval arithmetic semantics, enabling efficient sandbox execution. Here, we present our translation to interval arithmetic and prove it sound in Isabelle/HOL.

The major design choice here is arithmetic representation. We wish to keep compilation simple, support a wide variety of hardware, and keep the arithmetic soundness proof simple. We chose fixed-precision integer (interval) arithmetic because it is widely used in embedded software for its predictability and is universally supported by hardware and compilers. In Section 6, we show that limited precision was not an issue on our hardware platform, because physics limits the range and precision of sensor values.

Semantics. The transition to interval arithmetic does not require transforming the program source; we merely assign a new semantics to the existing constructs of dL. Representative cases of the term, formula, and program semantics are in Fig. 5. We write ω_I, ν_I for *interval states* assigning to each variable x an interval $[l,u]$ of 32-bit machine words for lower and upper bounds on the (real number) value of x, respectively. Machine words are interpreted as signed integers in standard two's-complement format, excepting sentinel values for negative (∞_w^-) and positive (∞_w^+) infinity.

We write $\omega_I[\![\theta]\!] : [\overline{\mathbb{R}}, \overline{\mathbb{R}}]$ for the value of term θ in the interval state ω_I, which is a closed interval in the extended reals. Likewise, we write $(\omega_I, \nu_I) \in [\![\alpha]\!]$ when interval state ω_I can reach ν_I upon running HP α. Because interval arithmetic is conservative, the resulting formula semantics is three-valued: we write $\omega_I[\![\phi]\!] = \top$ when ϕ is definitely true in interval state ω_I, $\bot$ when it is definitely false, or U when it is unknown. Arithmetic operations augmented with overflow checks are written with a subscript $_w$ and $\mathrm{trunc}(w)$ returns positive or negative infinity when w is out of range.

We implement bounds checking by sign-extending to 64-bit words, where our operations on 32-bit values are guaranteed not to overflow, and then checking the result. This is done, e.g., in $\hat{+}_w$ and $\check{+}_w$, (casts between 32- and 64-bit words are omitted for brevity). Rounding modes differ in handling of infinite inputs, e.g., $\infty_w^- + \infty_w^+$ is indeterminate, bounded below only by ∞_w^-, and bounded above only by ∞_w^+.

In three-valued semantics, inequalities $<_w, \leq_w$ have the truth value U (unknown) when the intervals for both sides of an inequality overlap. For example, $x <_w x \check{+}_w y$ could be either true or false in the state $\nu_I = \{x \mapsto [1,2], y \mapsto [0,1]\}$, so conservatively is U. In contrast, $x \leq_w x \check{+}_w y$ is $\top$.

$$\mathrm{trunc}(w) = \max(\infty_w^-, \min(\infty_w^+, w))$$

$$w_1 \hat{+}_w w_2 = \textbf{if } \max(w_1,w_2) \in \{\infty_w^+, \infty_w^-\} \textbf{ then } \max(w_1,w_2) \textbf{ else } \mathrm{trunc}(w_1 + w_2)$$

$$w_1 \check{+}_w w_2 = \textbf{if } \min(w_1,w_2) \in \{\infty_w^+, \infty_w^-\} \textbf{ then } \min(w_1,w_2) \textbf{ else } \mathrm{trunc}(w_1 + w_2)$$

$$\omega_I[\![(\theta_1 + \theta_2)]\!] = [l_1 \check{+}_w l_2, u_1 \hat{+}_w u_2] \text{ where } \omega_I[\![(\theta_i)]\!] = [l_i, u_i]$$

$$\omega_I[\![(\theta_1 < \theta_2)]\!] = \begin{cases} \top & \text{if } \omega_I[\![(\theta_i)]\!] = (l_i,u_i) \text{ and } u_1 < l_2 \\ \bot & \text{if } \omega_I[\![(\theta_i)]\!] = (l_i,u_i) \text{ and } l_1 \geq u_2 \\ U & \text{otherwise} \end{cases}$$

$$(\omega_I, \nu_I) \in [\![(x := \theta)]\!] \text{ iff } \nu_I = \omega_I \text{ except } \nu_I(x) = \omega_I[\![(\theta)]\!]$$

$$(\omega_I, \nu_I) \in [\![(?Q)]\!] \quad \text{iff } \omega_I = \nu_I \text{ and } \omega_I[\![(Q)]\!] = \top$$

$$(\omega_I, \nu_I) \in [\![(\alpha \cup \beta)]\!] \text{ iff } (\omega_I, \nu_I) \in [\![(\alpha)]\!] \text{ or } (\omega_I, \nu_I) \in [\![(\beta)]\!]$$

$\wedge$	$\top$	U	$\bot$
$\top$	$\top$	U	$\bot$
U	U	U	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$

$\vee$	$\top$	U	$\bot$
$\top$	$\top$	$\top$	$\top$
U	$\top$	U	U
$\bot$	$\top$	U	$\bot$

Figure 5. Interval arithmetic for executable dL, 3-valued truth tables (true $\top$, false $\bot$, unknown U)

Relating Real and Interval Semantics. We now formalize our notion of correctness: the interval semantics is sound with respect to real-number semantics if all word intervals contain their corresponding real numbers. Formally, we define a notation $\nu \in [\![(\nu_I)]\!]$ saying the values of all variables in interval state ν_I contain their correspondents in real-number state ν. We define the notation $r \in [\![(w_l, w_u)]\!]$ likewise when the real number r is in the interval $[w_l, w_u]$:

$$r \in [\![(w_l, w_u)]\!] \text{ iff } w_l \overset{\text{wr}}{\leq} r \text{ and } w_u \overset{\text{wr}}{\geq} r$$

$$\omega \in [\![(\omega_I)]\!] \text{ iff } \forall x \in \mathcal{V}\ \ \omega(x) \in [\![(\omega_I(x))]\!]$$

To simplify the proof structure, the definition is decomposed into one-sided safe bounds $w \overset{\text{wr}}{\leq} r$ and $w \overset{\text{wr}}{\geq} r$ saying word w is a lower or upper bound for real r, respectively:

$$w \overset{\text{wr}}{\leq} r \text{ iff } w \overset{\text{wr}}{=} r' \text{ for some } r' \leq r$$

$$w \overset{\text{wr}}{\geq} r \text{ iff } w \overset{\text{wr}}{=} r' \text{ for some } r' \geq r$$

The inexact bounds are defined with exact bounds $w \overset{\text{wr}}{=} r$ saying word w exactly represents real r. Here, `w2r` is the standard injection of two's-complement words into reals:

$$\infty_w^+ \overset{\text{wr}}{=} r \text{ iff } r \geq \texttt{w2r}(\infty_w^+)$$

$$\infty_w^- \overset{\text{wr}}{=} r \text{ iff } r \leq \texttt{w2r}(\infty_w^-)$$

$$w \overset{\text{wr}}{=} \texttt{w2r}(w) \text{ otherwise}$$

Soundness. We use the above definitions to state and prove soundness theorems that conservatively relate the interval semantics to the real semantics.

Theorem 6 (Soundness for terms)**.** *Interval valuations of terms contain their real valuations. That is, if* $\omega[\![\theta]\!] = r$ *and* $\omega \in [(\omega_I)]$ *then* $r \in \omega_I[(\theta)]$.

Proof. By induction on θ. We examine the representative case $\theta = \theta_1 + \theta_2$. The case reduces to qualitative correctness of rounding modes: upward-rounding operations preserve upper bounds and downward-rounding operations preserve lower bounds. We prove these properties ourselves as they are not provided by existing Isabelle/HOL arithmetic libraries [46]. For example, we prove: If $w_1 \overset{\text{wr}}{\geq} r_1$ and $w_2 \overset{\text{wr}}{\geq} r_2$ then $w_1 \hat{+}_w w_2 \overset{\text{wr}}{\geq} r_1 + r_2$. The proof is by cases, following the structure of the definition of $w_1 \hat{+}_w w_2$. When bound checks detect overflow, they soundly return infinities. When w_1 and w_2 are both finite and bounds checks pass, it suffices to show that the casts are sound, which they are. □

Theorem 7 (Soundness for formulas)**.** *If the interval semantics of a formula is true or false, the real semantics agree.*

- *If* $\omega_I[(\phi)] = \top$ *and* $\omega \in [(\omega_I)]$ *then* $\omega \models \phi$.
- *If* $\omega_I[(\phi)] = \bot$ *and* $\omega \in [(\omega_I)]$ *then* $\omega \not\models \phi$.
- *If* $\omega_I[(\phi)] = U$ *we make no claim.*

Proof. By induction on fact $\omega_I \in [(\phi)]$, appealing to Theorem 6. We show the representative case $\phi \equiv (\theta_1 < \theta_2)$. Comparisons $\theta_1 < \theta_2$ bridge terms to formulas. For soundness, we conservatively compare the *upper* bound of θ_1 with the *lower* bound of θ_2, and comparison of overlapping intervals returns undefined (U). Stated formally: If $w_1 \overset{\text{wr}}{\geq} r_1$ and $w_2 \overset{\text{wr}}{\leq} r_2$ and $w_1 <_w w_2$ then $r_1 < r_2$. The proof is direct, by the definitions of $\overset{\text{wr}}{\leq}, \overset{\text{wr}}{\geq}, \overset{\text{wr}}{=}$, and $<_w$. □

Theorem 8 (Soundness for programs)**.** *If* $(\omega_I, \nu_I) \in [(\alpha)]$ *and* $\omega \in [(\omega_I)]$*, then there exists* $\nu \in [(\nu_I)]$ *where* $(\omega, \nu) \in [\![\alpha]\!]$.

Proof. By induction on programs α, using Theorem 7. □

Together, these show that all program behaviors accepted by the sandbox program in interval semantics correspond to behavior in the real semantics, which is safe by Theorem 4:

Corollary 9 (Sandbox soundness)**.** *If* $(\text{ctrl}; \text{plant})^*$ *is verified and sensing is sound* ($\omega \in [(\omega_I)]$) *and the sandbox implementation transitions* ($\omega_I[(\phi)] = \top$ *and* $(\omega_I, \nu_I) \in [(\alpha)]$)*, then there is a real state* ν *underlying the final interval state* ($\nu \in [(\nu_I)]$) *which is safe, i.e.,* $\nu \models \psi$.

Proof. By sensing soundness and Theorem 7, $\omega \models \psi$. By Theorem 4, since $(\text{ctrl}; \text{plant})^*$ is verified then $\phi \rightarrow [\text{sandbox}]\psi$ is valid. By Theorem 8, $(\omega, \nu) \in [\![\alpha]\!]$ for some $\nu \in [(\nu_I)]$. By the above implication, $\nu \models \psi$. □

Now the sandbox is executable at a high level and still safe. Next, we will soundly implement the executable interval semantics at the machine level.

5 Sandbox Implementation in CakeML

By Corollary 9, we may now understand the sandbox HP from Fig. 2 using the interval semantics. This allows us to implement the monitoring checks (10) and (14) using machine arithmetic. However, the sandbox still contains high-level abstract constructs. Nondeterministic assignments are used to represent sensing in (8) and (13) and external control in (9). Sound implementation of external control (9) and actuation (12) are still expressed abstractly. Nondeterminism must be resolved also in choosing between external (10) and fallback (11) control, and in choosing the sandbox loop duration.

In this section, we explain how the sandbox is implemented as a CakeML program (Section 5.1). We resolve the aforementioned sources of nondeterminism, external controller calls, and actuators all with CakeML's support for *foreign function interfaces* (*FFIs*). The resulting program is then compiled down to machine code (ARMv6, x64, etc.) using the verified CakeML compiler [45].

By employing the verified CakeML compiler, we know that the compiled machine code soundly implements CakeML source programs. It remains to show (Section 5.3) that our CakeML program soundly implements the sandbox. This verification step is made easier because CakeML is itself a high-level programming language with an accompanying suite of verification tools [16, 35]. The CakeML program, however, senses and actuates in the real world, so its soundness relies on assumptions about the correctness of sensor and actuator FFIs (Section 5.2).

5.1 CakeML Sandbox

We first explain how we implement nondeterminism and external interaction. The following pseudocode snippet illustrates the sandbox loop implementation. Corresponding lines from Fig. 2 are indicated on the right.

```
fun cmlSandboxBody state =
  if not (stop ()) then
    state.ctrl+:= extCtrl state;                          (9)
    state.ctrl := if intervalSem ctrlMon state = ⊤
                  then state.ctrl+                        (10)
                  else fallback state;                    (11)
    actuate state.ctrl;                                   (12)
    state.sensors+:= sense ();                            (13)
    if intervalSem plantMon state = ⊤ then                (14)
      Runtime.fullGC ();
      state.sensors := state.sensors+;                    (15)
      cmlSandboxBody state                                (16)
    else violation "Plant Violation"
```

The tail-recursive function `cmlSandboxBody` keeps track of a CakeML representation of the current state (`state`). We use record-update notation for `state`, closely following the assignments in Fig. 2. Loop termination is decided by the `stop` FFI wrapper. The `stop` wrapper itself makes an FFI call to external code (`ffiStop`) which decides whether to stop the loop, e.g., upon user request or battery depletion.

Nondeterministic assignments for external control and actuation are implemented with the `extCtrl` and `sense` FFI wrappers which control and sense via external drivers. From the current state variable vector $\vec{x}$, we single out the sensor variables (`state.sensors`), actuated variables (`state.ctrl`), and constants (`state.consts`); $\vec{x}^+$ is treated likewise.

The `actuate` FFI wrapper executes the control decision `state.ctrl`, taken from `extCtrl` when the controller monitor `ctrlMon` is satisfied or the `fallback` otherwise.

The above nondeterminism came from the environment and was thus resolved externally with FFIs. The nondeterministic choice between (10) and (11), in contrast, simply provides us freedom in controller implementation. We exploit this freedom when the ternary truth-value of `ctrlMon` in the interval semantics (`intervalSem`) is unknown (U): we are free to use the fallback (11) even when `ctrlMon` is not definitely false ($\perp$), so we conservatively use it in the unknown (U) case as well.

If the *plant* monitor fails, however, sandboxing cannot guarantee safety. Here, `cmlSandboxBody` exits the control loop by calling the `violation` function with an error message. The function returns control to user code, which may initiate (unverified) best-effort recovery measures in the case of plant violations. For time-triggered controllers, the plant monitor only holds when the system delay is within our specified limit ε. We minimize the risk of a delay violation by garbage-collecting (`Runtime.fullGC`) after each cycle, making runtimes predictable. The cost is negligible in practice because each control cycle does minimal heap allocation.

The sandbox entry point `cmlSandbox`, elided here, simply invokes `cmlSandboxBody` on the initial state after checking initial conditions ϕ. We have thus reduced implementation of the sandbox to implementation of the FFIs.

5.2 CakeML FFIs

We now specify and implement the FFIs. FFIs bridge CakeML to the external world: physical sensing/actuation and untrusted control. Thus, the crucial specification step is to model external behavior in HOL4: We write *es*`:ext` for an *external state*, a record capturing all external state and effects, current and future. The external state is taken as the ground truth of the world, which all (e.g. sensing/actuation) FFIs must read or change. This makes the notion of sensing/actuation correctness precise. The 6 FFIs assumed by VeriPhy are summarized in Table 2, along with their informal meanings. Of these, we take `ffiSense` and `ffiStop` as our examples.

Table 2. External functions and their intended meaning

External func.	Intended Meaning
`ffiConst`	Get the values of system constants
`ffiSense`	Get the current sensor readings
`ffiExtCtrl`	Get the next (untrusted) control decision
`ffiActuate`	Actuate a control decision
`ffiStop`	Check whether to run more control cycles
`ffiViolation`	Exit control loop due to a fatal violation

FFI Model. Each FFI specification consults the external state to determine the ground truth of the current state and effect of external code. They each return a result `r` and a new external state *es* when invoked safely (i.e. `SOME(r,es')`) or `NONE` if calling conventions were violated. For example, the `ffiSense` calling convention expects one word per sensor in the array *bytes*, then we specify that the values sensed by `ffiSense` match the ground truth *es*`.sensor_vals`:

```
ffiSense bytes (es:ext) =
  if LEN bytes = NSENSORS*WORD ∧
     LEN es.sensor_vals = LEN bytes
  then SOME (word_to_bytes es.sensor_vals,es)
  else NONE
```

Unlike `ffiSense`, which must always return the current sensor values, `ffiStop` has complete freedom to decide when the loop should stop. This external nondeterminism is resolved by querying an oracle (*es*`.stop_oracle`):

```
ffiStop bytes (es:ext) =
  if LEN bytes = 1
  then SOME (query es.stop_oracle,es)
  else NONE
```

When queried, the oracle returns a bit, with 1 indicating that the sandbox loop should stop.

Neither `ffiSense` nor `ffiStop` modifies the state of the external world, so the external state *es* is unchanged. The external state is modified, e.g., when `ffiActuate` sends control values to actuators.

The full specification is ≈150 lines of HOL4, and formally captures the assumed behavior of each FFI from Table 2.

FFI Stub Implementation. The end-user must provide external FFI implementations. Here, we implement `ffiSense` by filling a VeriPhy-generated C stub with calls to application-specific drivers. Per the specification, `ffiSense` populates `sensor_vals` with the actual sensor values:

```
void ffiSense(int32_t *sensor_vals, long nSensors) {
  sensor_vals[0] = distanceDriver();  // return d
  senser_vals[1] = currentTime();     // return t
}
```

CakeML FFI Wrapper. The CakeML sandbox program accesses the FFIs through CakeML wrapper functions. For example, the `sense` function wraps the `ffiSense` FFI:

```
fun sense () =
let val sensorArr = Word8Array.array (NSENSORS*WORD) 0
    val () = #(ffiSense) sensorArr
in arr_to_list sensorArr end
```

The function first allocates a byte array sensorArr with one word per sensor value. It then invokes ffiSense using CakeML's FFI call syntax #(ffiSense). Once sensorArr is populated with real sensor values, sense returns them reformatted as a list.

The remaining FFIs are modeled and implemented similarly to the representative examples shown above, with ffiCtrl and ffiActuate also having their own oracles to model external control and actuation, respectively.

5.3 Verifying the CakeML Sandbox

Next, we verify the CakeML program cmlSandbox, assuming that the FFIs behave according to our FFI model. The main verification work is carried out with CakeML's Characteristic Formulae (CF) framework [16], which allows reasoning about the FFIs with separation logic-style assertions. As with interval arithmetic soundness, our results are generic across all sandbox instances.

We write $[\![\omega]\!]$ for a CakeML state containing an external state es:ext and a runtime store. We write $\left([\![\omega]\!],[\![\nu]\!]\right) \in [\![\texttt{cmlSandbox}]\!]$ to mean that executing the CakeML sandbox (cmlSandbox) from the initial CakeML state $[\![\omega]\!]$ terminates with the CakeML state $[\![\nu]\!]$, see [16] for formal details. The states implicitly agree with cmlSandbox as to which variables are sensors/actuators, etc. For any CakeML state $[\![\omega]\!]$, the corresponding interval state $[\![\omega]\!]$ represents each value $[\![\omega]\!](x) = w$ exactly by a point-interval $[w,w]$, which implies that sensing is exact. Sensor uncertainty could in principle be encoded with non-point intervals.

Theorem 10 (CakeML sandbox correctness)**.** *For any initial CakeML state $[\![\omega]\!]$, assuming that its* stop *oracle eventually stops the loop (by returning the bit* 1 *when queried), then we have a CakeML state $[\![\nu]\!]$ such that $\left([\![\omega]\!],[\![\nu]\!]\right) \in [\![\texttt{cmlSandbox}]\!]$. In addition,*

1. *If $[\![\omega]\!]$ violates initial condition ϕ, then* cmlSandbox *leaves the initial CakeML state unchanged: $[\![\omega]\!] = [\![\nu]\!]$.*
2. *Otherwise, either the* stop *oracle of $[\![\nu]\!]$ stopped the loop when queried and $([\![\omega]\!],[\![\nu]\!]) \in [\![\text{(sandbox)}]\!]$, or*
3. *There exists $[\![\mu]\!]$ where $([\![\omega]\!],[\![\mu]\!]) \in [\![\text{(sandbox)}]\!]$ and $[\![\nu]\!]$ was obtained by actuating (12) in $[\![\mu]\!]$ where (after sensing)* intervalSem *raises a* plantMon *(14) violation.*

We assume that the stop oracle eventually stops the loop because real systems do not run forever. Under this assumption, soundness is verified simply by induction on execution traces. The violation in Case 3 of Theorem 10 is raised conservatively when plantMon has an unknown truth value (U), analogously to the control monitor ctrlMon. The violation is guaranteed to be raised when plantMon *first* fails, ensuring early detection of any model deviations.

Using CakeML's compiler correctness theorem [45], we extend Theorem 10 to the compiled, machine code implementation of cmlSandbox. We write CML(cmlSandbox) for the result of running the CakeML compiler on cmlSandbox, $\{\!|\texttt{CML(cmlSandbox)}|\!\}$ for its (machine code) semantics, and accordingly $\{\!|\omega|\!\}$ for a machine-level program state.

Theorem 11 (Sandbox machine code correctness)**.** *Under the standard CakeML compiler correctness assumptions [45], let $\{\!|\omega|\!\}$ be an initial machine state whose* stop *oracle eventually stops the loop. Then we have a machine state $\{\!|\nu|\!\}$ such that $\left(\{\!|\omega|\!\},\{\!|\nu|\!\}\right) \in \{\!|\texttt{CML(cmlSandbox)}|\!\}$, and $\{\!|\omega|\!\},\{\!|\nu|\!\}$ satisfy one of the three cases listed in the conclusion of Theorem 10. The machine code may also exit with an out-of-memory error if the CakeML runtime exhausts its heap or stack.*

As a corollary, we have the following end-to-end chain of correctness guarantees for VeriPhy:

Corollary 12 (End-to-end implementation guarantees)**.** *Under the assumptions of Theorem 11, assume further that Case 2 of the theorem occurs and the CakeML runtime does not run out of memory. Let* (ctrl; plant)* *be verified, external interaction be sound, then there is a real state ν underlying state $\{\!|\nu|\!\}$ which is safe, i.e., $\nu \models \psi$.*

Proof. By composing Corollary 9 with Case 2 of Theorem 11. Here soundness of external interactions consists of sensing soundness per Corollary 9 and correctness of FFI with respect to the external state model and the CakeML compiler correctness assumptions [45]. □

6 Experimental Evaluation

The pipeline has successfully synthesized sandboxes for our velocity-controlled robot example, a train safety controller [41], and acceleration-controlled motion [43]. Our proofs guarantee soundness, so sandbox controllers execute only provably safe control choices. We experimentally evaluate *operational suitability* of the velocity-controlled robot, showing that sandboxing does not make the controller overly conservative. We see that the controllers and machine arithmetic are fast and precise enough in practice and that monitors alert the user if modeling simplifications affect safety, but do not raise excessive false alarms. Since monitors detect model violations, the absence of excessive alarms also helps validate the input model. We run our robot example on commodity robot hardware with several controllers. This common platform minimizes hardware cost while our multiple controls allow testing the approach's generality. We augment the experiments with simulations showing how the sandbox responds to non-compliant environments.

Hardware Platform and Calibration. We use a GoPiGo3 Raspberry Pi-based robot. It is equipped with two separately

controlled motors and a laser distance sensor with 25° field-of-view and typical indoor measurement range on white background of 200 cm with ≈94% obstacle detection rate. Depending on operating temperature and voltage, the distance measurements are off by at most ±3 cm. The motors take speed commands in the range $-25\,\frac{\text{cm}}{\text{s}}$ to $25\,\frac{\text{cm}}{\text{s}}$, controlled internally with a proportional-integral-derivative (PID) controller. It has a stopping margin of about 2.5 cm from engaging "brakes" by setting $v = 0$ until full stop from maximum speed. The sensed distance incorporates this margin, closely mimicking instantaneous stopping per our model.

Drivers. GoPiGo3 provides C drivers for the motors, but only Python drivers for the distance sensor. Between sensing, control, and actuation, the control cycle time is 180–220 ms. It is dominated by distance sensor interaction through Python.

Experiment Setup. The robot is initially stationary, 75 cm from an obstacle, then drives straight toward the obstacle with user-defined constant speeds $10, 15, 20, 25\,\frac{\text{cm}}{\text{s}}$ (maximum speed of the robot). Since the robot measures time in ms, we measure speed in $\frac{\text{mm}}{\text{s}}$ and distance in μm. Thus the greatest distance in the system, 75 cm, can be represented in 20 bits, well within our 32-bit limit. We performed the experiment with both stationary and moving obstacles. The robot stops close to a stationary obstacle, with ≈3 cm safety margin to account for sensor and actuator uncertainty. If the obstacle moves away, the robot follows, stopping close to the obstacle's final position. If the obstacle moves closer, the robot stops before reaching the obstacle.

We tested two implementations of the untrusted external controller. Controller A follows a user-defined speed when safe and otherwise stops. This is safe and thus does not violate the monitor. Controller B first sets a user-defined speed then spikes to maximum speed near the obstacle. This is unsafe and violates the monitor, invoking fallback control. Our experiments on both the real robot and a simulated plant record distance, speed, and monitor violations vs. time.

Experiment Results. Fig. 6 plots distance over time in simulation for maximum speed $V = 25\frac{\text{cm}}{\text{s}}$, system delay $\varepsilon = 220$ ms, initial distance 75 cm, with varying sensor disturbance and both passive and malicious obstacles. Fig. 7 plots results on the real robot. Both figures show that monitors correctly detect plant violations. Fallback is then engaged as a safety best-effort, which ensured safety in simulation.

The robot engaged fallback at ≈4.6 cm from the obstacle, stopping at ≈2.6 cm (due to the safety margin). Under small disturbances, the plant monitor holds and safety is assured. Malicious obstacles or dangerously high disturbances are detected as plant violations, triggering fallback. We simulated controller faults at $d \approx 50$ cm by issuing $v = V = 25\frac{\text{cm}}{\text{s}}$ continually. The fallback engages right before the faulty controller would become unsafe ($d < \varepsilon V$).

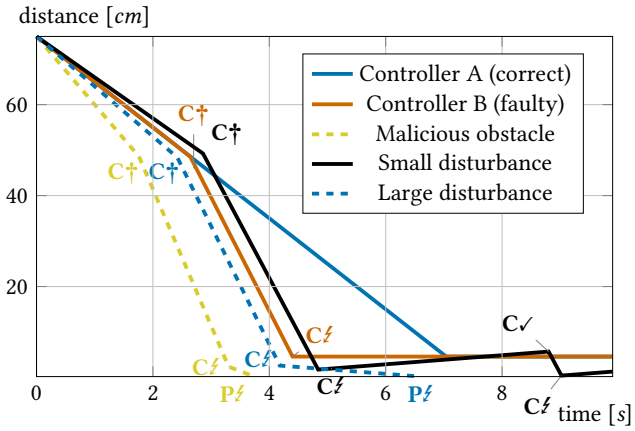

Figure 6. Controller sandbox, simulated plant. Solid lines: sandbox controller safe; fallback engaged if control violation occurs. Dashed lines: plant violation, environment caused collision. **C†**,**C↯**,**P↯**,**C✓**indicate speed spike, control/plant violations, and restoration of normal control, respectively.

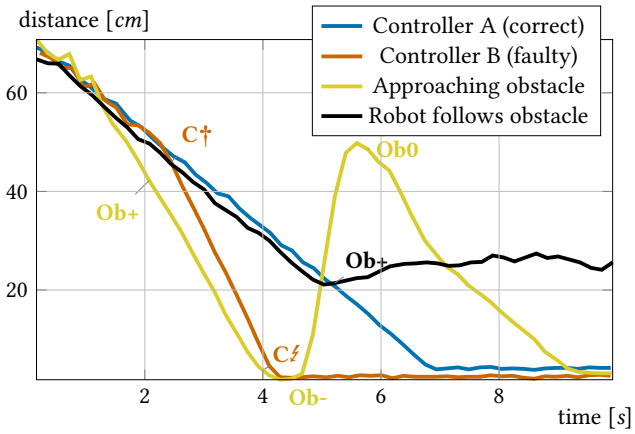

Figure 7. Controller sandbox, real robot: correct controller approaching a stationary obstacle, faulty controller approaching a stationary obstacle, obstacle approaches robot, and robot follows a moving obstacle. **Ob+**, **Ob0**, **Ob-** indicate proceeding, stopped, and receding obstacles, respectively.

7 Related Work

CPS Verification. Differential dynamic logic [37, 38], as implemented in KeYmaera X [15], has been successfully applied in a number of case studies [39]. Soundness of its core calculus [40] has been cross-verified in Isabelle/HOL and Coq [2], which this work extended to a verified proof-term checker in Isabelle/HOL.

General-purpose logics have also been used for CPS proofs. ROSCoq [1] and VeriDrone [27] allow CPS specification, implementation, verification, and code generation in Coq.

However, they do not synthesize and automatically verify monitors nor is their machine code verified.

We provide greater automation: the user need only verify the source model and can exploit KeYmaera X automation. On the other hand, Coq provides ROSCoq and VeriDrone with greater freedom in models and verified controllers, which VeriPhy has only in the untrusted controller.

Hybrid model checking [8, 10, 13] typically has a large trusted base and sacrifices expressiveness to increase automation, being restricted, e.g., to linear differential equations, bounded-time safety, or bounded state-space verification.

Toolchains. High-Assurance SPIRAL [12] translates ModelPlex monitors to x64 machine code. Its verification is not end-to-end: because it only treats hybrid systems syntactically, it cannot verify across semantic gaps, e.g., in interval arithmetic and compilation. Its optimizer employs features such as SIMD instructions that would make verified compilation especially challenging. Ivory [11] and Tower [36] are used to ensure high-assurance embedded software is memory safe, but do not verify the functional correctness of cyber nor physical behavior.

Verified Compilation. We use the CakeML [45] verified ML compiler and its associated verification tools [16, 35]. CakeML is higher-level than other languages such as Clight with verified compilers such as floating-point CompCert [4]. This makes the verification of sandbox implementation in CakeML against hybrid systems semantics painless. We chose this over, e.g., translation validation with unverified compilers [44] since translation validation can be brittle.

Lustre [17] is a CPS-centric language with a verified compiler, Vélus [7]. Writing and compiling a Lustre controller provides no end-to-end guarantees because physical modeling and verification are left unanswered. It could be used as a code generation target, but this would be a detour because the Lustre language differs greatly from hybrid programs.

Machine Arithmetic Verification. Machine arithmetic correctness is a major VeriPhy component. We verify arithmetic soundness foundationally. This is an active research area with libraries available in HOL Light [18], Coq [3, 5, 9, 30], Isabelle/HOL [46], etc. Of these, only PFF in Coq [9] provides the qualitative rounding correctness results we need, so we prove them in Isabelle/HOL using the seL4 [24] machine word library. We chose Isabelle/HOL and HOL4 over Coq because their combination of cutting-edge analysis libraries [23], mature formalization of dL [2], proof-producing code extraction [35], and classical foundations positions them well for our end-to-end pipeline. Static analysis of arithmetic for hybrid systems has been studied [6, 26, 28], but without foundational safety proofs for general dynamics.

8 Conclusions and Future Work

VeriPhy is the first automatic, verified pipeline from verified CPS models to verified controller executables. High-level dL proofs provide safe interaction between code and physics, which we have transported to the implementation level.

We summarize the chain of proofs in Fig. 8: A verified dL model is transformed to a verified sandbox featuring a synthesized monitor. The sandbox is then soundly reinterpreted over interval arithmetic and compiled by CakeML to, e.g., ARM or x64. So long as the executable does not report a model violation, the system remains in a safe state.

$$
\begin{array}{rl}
\nu \models \psi & \\
\Uparrow & \text{by Section 3: Theorem 4} \\
(\omega,\nu) \in \underbrace{[\![\texttt{sandbox}]\!]}_{\text{dL (KeYmaera X)}} & \text{Real arithmetic, nondeterministic} \\
\Uparrow & \text{by Section 4: Theorem 8} \\
\left(\omega_I,\nu_I\right) \in \underbrace{[\texttt{(sandbox)}]}_{\text{dL (Isabelle/HOL)}} & \text{Interval word arithmetic, nondeterministic} \\
\Uparrow & \text{by Section 5: Theorem 10} \\
\left([\{\omega\}],[\{\nu\}]\right) \in \underbrace{[\{\texttt{cmlSandbox}\}]}_{\text{CakeML (HOL4)}} & \text{Interval word arithmetic, deterministic} \\
\Uparrow & \text{by CakeML compiler [45]} \\
\left(\{\!|\omega|\!\},\{\!|\nu|\!\}\right) \in \underbrace{\{\!|\texttt{CML(cmlSandbox)}|\!\}}_{\text{ARM/x64}} & \text{Interval word arithmetic, machine-executable}
\end{array}
$$

Figure 8. End-to-end proof chain for end-to-end result

To cross the wide gap between hybrid systems and executable code, VeriPhy employs multiple tools, gaining convenience at the cost of an enlarged trusted base. The VeriPhy approach incorporates measures to ensure that the connections between tools are sound, which we wish to strengthen in future work to further mitigate trusted base size. The cross-verification of KeYmaera X into Isabelle/HOL can be made more trustworthy by incorporating proof-producing arithmetic solvers [19, 29, 33, 42]. The link between Isabelle/HOL and HOL4 could be made more trustworthy by automatically converting specifications with OpenTheory [22]. The treatment of memory-management, while already trustworthy, could be made even more suitable for real-time systems by generating allocation-free code in the backend. With these additions, we can maintain convenient end-to-end guarantees from high-level models with even higher reliability.

Acknowledgments. This material is based upon work supported by the National Science Foundation under NSF CAREER Award CNS-1054246, by the AFOSR under grant number FA9550-16-1-0288, and by DARPA under agreement number FA8750-12-2-0291. The first and second authors were also supported by the Department of Defense through the National Defense Science & Engineering Graduate Fellowship Program, and by A*STAR, Singapore, respectively.

References

[1] Abhishek Anand and Ross A. Knepper. 2015. ROSCoq: Robots powered by constructive reals. In *ITP (LNCS)*, Christian Urban and Xingyuan Zhang (Eds.), Vol. 9236. Springer, 34–50. https://doi.org/10.1007/978-3-319-22102-1_3

[2] Brandon Bohrer, Vincent Rahli, Ivana Vukotic, Marcus Völp, and André Platzer. 2017. Formally verified differential dynamic logic. In *Certified Programs and Proofs - 6th ACM SIGPLAN Conference, CPP 2017, Paris, France, January 16-17, 2017*, Yves Bertot and Viktor Vafeiadis (Eds.). ACM, 208–221. https://doi.org/10.1145/3018610.3018616

[3] Sylvie Boldo, Jean-Christophe Filliâtre, and Guillaume Melquiond. 2009. Combining Coq and Gappa for certifying floating-point programs. In *MKM, Held as Part of CICM (LNCS)*, Jacques Carette, Lucas Dixon, Claudio Sacerdoti Coen, and Stephen M. Watt (Eds.), Vol. 5625. Springer, 59–74. https://doi.org/10.1007/978-3-642-02614-0_10

[4] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. 2013. A formally-verified C compiler supporting floating-point arithmetic. In *ARITH*, Alberto Nannarelli, Peter-Michael Seidel, and Ping Tak Peter Tang (Eds.). IEEE Computer Society, 107–115. https://doi.org/10.1109/ARITH.2013.30

[5] Sylvie Boldo and Guillaume Melquiond. 2011. Flocq: A unified library for proving floating-point algorithms in Coq. In *ARITH*, Elisardo Antelo, David Hough, and Paolo Ienne (Eds.). IEEE Computer Society, 243–252. https://doi.org/10.1109/ARITH.2011.40

[6] Olivier Bouissou, Eric Goubault, Sylvie Putot, Karim Tekkal, and Franck Védrine. 2009. HybridFluctuat: A static analyzer of numerical programs within a continuous environment *(LNCS)*, Ahmed Bouajjani and Oded Maler (Eds.), Vol. 5643. Springer, 620–626. https://doi.org/10.1007/978-3-642-02658-4_46

[7] Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. 2017. A formally verified compiler for Lustre. In *PLDI*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 586–601. https://doi.org/10.1145/3062341.3062358

[8] Xin Chen, Erika Ábrahám, and Sriram Sankaranarayanan. 2013. Flow*: An analyzer for non-linear hybrid systems. In *CAV (LNCS)*, Natasha Sharygina and Helmut Veith (Eds.), Vol. 8044. Springer, 258–263. https://doi.org/10.1007/978-3-642-39799-8_18

[9] Marc Daumas, Laurence Rideau, and Laurent Théry. 2001. A generic library for floating-point numbers and its application to exact computing. In *TPHOLs (LNCS)*, Richard J. Boulton and Paul B. Jackson (Eds.), Vol. 2152. Springer, 169–184. https://doi.org/10.1007/3-540-44755-5_13

[10] Parasara Sridhar Duggirala, Sayan Mitra, Mahesh Viswanathan, and Matthew Potok. 2015. C2E2: A verification tool for stateflow models. In *TACAS (LNCS)*, Christel Baier and Cesare Tinelli (Eds.), Vol. 9035. Springer, 68–82. https://doi.org/10.1007/978-3-662-46681-0_5

[11] Trevor Elliott, Lee Pike, Simon Winwood, Patrick C. Hickey, James Bielman, Jamey Sharp, Eric L. Seidel, and John Launchbury. 2015. Guilt free ivory. In *Haskell*. 189–200. https://doi.org/10.1145/2804302.2804318

[12] Franz Franchetti, Tze Meng Low, Stefan Mitsch, Juan Paolo Mendoza, Liangyan Gui, Amarin Phaosawasdi, David Padua, Soummya Kar, José M. F. Moura, Mike Franusich, Jeremy Johnson, André Platzer, and Manuela Veloso. 2017. High-assurance SPIRAL: End-to-end guarantees for robot and car control. *IEEE Control Systems* 37, 2 (2017), 82–103. https://doi.org/10.1109/MCS.2016.2643244

[13] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. 2011. SpaceEx: Scalable verification of hybrid systems. In *CAV (LNCS)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.), Vol. 6806. 379–395. https://doi.org/10.1007/978-3-642-22110-1_30

[14] Nathan Fulton, Stefan Mitsch, Brandon Bohrer, and André Platzer. 2017. Bellerophon: Tactical theorem proving for hybrid systems. In *ITP (LNCS)*, Mauricio Ayala-Rincón and César A. Muñoz (Eds.), Vol. 10499. Springer, 207–224. https://doi.org/10.1007/978-3-319-66107-0_14

[15] Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völp, and André Platzer. 2015. KeYmaera X: An axiomatic tactical theorem prover for hybrid systems. In *CADE (LNCS)*, Amy Felty and Aart Middeldorp (Eds.), Vol. 9195. Springer, 527–538. https://doi.org/10.1007/978-3-319-21401-6_36

[16] Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. 2017. Verified characteristic formulae for CakeML. In *ESOP (LNCS)*, Hongseok Yang (Ed.), Vol. 10201. Springer, 584–610. https://doi.org/10.1007/978-3-662-54434-1_22

[17] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. 1992. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Trans. Software Eng.* 18, 9 (1992), 785–793. https://doi.org/10.1109/32.159839

[18] John Harrison. 2006. Floating-point verification using theorem proving. In *Formal Methods for Hardware Verification, SFM (LNCS)*, Marco Bernardo and Alessandro Cimatti (Eds.), Vol. 3965. Springer, 211–242. https://doi.org/10.1007/11757283_8

[19] John Harrison. 2007. Verifying nonlinear real formulas via sums of squares. In *TPHOLs (LNCS)*, Klaus Schneider and Jens Brandt (Eds.), Vol. 4732. Springer, 102–118. https://doi.org/10.1007/978-3-540-74591-4_9

[20] Thomas A. Henzinger. 1996. The theory of hybrid automata. In *LICS*. IEEE Computer Society, 278–292. https://doi.org/10.1109/LICS.1996.561342

[21] Lars Hupel and Tobias Nipkow. 2018. A Verified Compiler from Isabelle/HOL to CakeML. In *ESOP (LNCS)*, Amal Ahmed (Ed.). Springer.

[22] Joe Hurd. 2011. The OpenTheory standard theory library. In *NFM (LNCS)*, Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.), Vol. 6617. Springer, 177–191. https://doi.org/10.1007/978-3-642-20398-5_14

[23] Fabian Immler and Christoph Traut. 2016. The flow of ODEs. In *ITP (LNCS)*, Jasmin Christian Blanchette and Stephan Merz (Eds.), Vol. 9807. Springer, 184–199. https://doi.org/10.1007/978-3-319-43144-4_12

[24] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2010. seL4: Formal verification of an operating-system kernel. *Commun. ACM* 53, 6 (2010), 107–115. https://doi.org/10.1145/1743546.1743574

[25] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A verified implementation of ML. In *POPL*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 179–192. https://doi.org/10.1145/2535838.2535841

[26] Rupak Majumdar, Indranil Saha, and Majid Zamani. 2012. Synthesis of minimal-error control software. In *EMSOFT*, Ahmed Jerraya, Luca P. Carloni, Florence Maraninchi, and John Regehr (Eds.). ACM, 123–132. https://doi.org/10.1145/2380356.2380380

[27] Gregory Malecha, Daniel Ricketts, Mario M. Alvarez, and Sorin Lerner. 2016. Towards foundational verification of cyber-physical systems. In *SOSCYPS@CPSWeek*. IEEE Computer Society, 1–5. https://doi.org/10.1109/SOSCYPS.2016.7580000

[28] Adolfo Anta Martinez, Rupak Majumdar, Indranil Saha, and Paulo Tabuada. 2010. Automatic verification of control system implementations. In *EMSOFT*, Luca P. Carloni and Stavros Tripakis (Eds.). ACM, 9–18. https://doi.org/10.1145/1879021.1879024

[29] Sean McLaughlin and John Harrison. 2005. A proof-producing decision procedure for real arithmetic. In *CADE (LNCS)*, Robert Nieuwenhuis (Ed.), Vol. 3632. Springer, 295–314. https://doi.org/10.1007/11532231_22

[30] Guillaume Melquiond. 2012. Floating-point arithmetic in the Coq system. *Inf. Comput.* 216 (2012), 14–23. https://doi.org/10.1016/j.ic.2011.09.005

[31] Stefan Mitsch and André Platzer. 2016. The KeYmaera X proof IDE: Concepts on usability in hybrid systems theorem proving. In *3rd Workshop on Formal Integrated Development Environment (EPTCS)*, Catherine Dubois, Dominique Mery, and Paolo Masci (Eds.), Vol. 240. Open Publishing Association, 67–81. https://doi.org/10.4204/EPTCS.240.5

[32] Stefan Mitsch and André Platzer. 2016. ModelPlex: Verified runtime validation of verified cyber-physical system models. *Form. Methods Syst. Des.* 49, 1 (2016), 33–74. https://doi.org/10.1007/s10703-016-0241-z Special issue of selected papers from RV'14.

[33] David Monniaux and Pierre Corbineau. 2011. On the generation of Positivstellensatz witnesses in degenerate cases. In *ITP (LNCS)*, Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk (Eds.), Vol. 6898. Springer, 249–264. https://doi.org/10.1007/978-3-642-22863-6_19

[34] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. 2012. RockSalt: better, faster, stronger SFI for the x86. In *PLDI*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 395–404. https://doi.org/10.1145/2254064.2254111

[35] Magnus O. Myreen and Scott Owens. 2012. Proof-producing synthesis of ML from higher-order logic. In *ICFP*, Peter Thiemann and Robby Bruce Findler (Eds.). ACM, 115–126. https://doi.org/10.1145/2364527.2364545

[36] Lee Pike, Patrick C. Hickey, James Bielman, Trevor Elliott, Thomas DuBuisson, and John Launchbury. 2014. Programming languages for high-assurance autonomous vehicles: Extended abstract. In *PLPV*. 1–2. https://doi.org/10.1145/2541568.2541570

[37] André Platzer. 2008. Differential dynamic logic for hybrid systems. *J. Autom. Reas.* 41, 2 (2008), 143–189. https://doi.org/10.1007/s10817-008-9103-8

[38] André Platzer. 2012. Logics of dynamical systems. In *LICS*. IEEE Computer Society, 13–24. https://doi.org/10.1109/LICS.2012.13

[39] André Platzer. 2016. Logic & proofs for cyber-physical systems. In *IJCAR (LNCS)*, Nicola Olivetti and Ashish Tiwari (Eds.), Vol. 9706. Springer, 15–21. https://doi.org/10.1007/978-3-319-40229-1_3

[40] André Platzer. 2017. A complete uniform substitution calculus for differential dynamic logic. *J. Autom. Reas.* 59, 2 (2017), 219–265. https://doi.org/10.1007/s10817-016-9385-1

[41] André Platzer and Jan-David Quesel. 2009. European Train Control System: A case study in formal verification. In *ICFEM (LNCS)*, Karin Breitman and Ana Cavalcanti (Eds.), Vol. 5885. Springer, 246–265. https://doi.org/10.1007/978-3-642-10373-5_13

[42] André Platzer, Jan-David Quesel, and Philipp Rümmer. 2009. Real world verification. In *CADE (LNCS)*, Renate A. Schmidt (Ed.), Vol. 5663. Springer, 485–501. https://doi.org/10.1007/978-3-642-02959-2_35

[43] Jan-David Quesel, Stefan Mitsch, Sarah Loos, Nikos Aréchiga, and André Platzer. 2016. How to model and prove hybrid systems with KeYmaera: A tutorial on safety. *STTT* 18, 1 (2016), 67–91. https://doi.org/10.1007/s10009-015-0367-0

[44] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. 2013. Translation validation for a verified OS kernel. In *PLDI*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 471–482. https://doi.org/10.1145/2462156.2462183

[45] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. 2016. A new verified compiler backend for CakeML. In *ICFP*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 60–73. https://doi.org/10.1145/2951913.2951924

[46] Lei Yu. 2013. A formal model of IEEE floating point arithmetic. *Archive of Formal Proofs* (2013). https://www.isa-afp.org/entries/IEEE_Floating_Point.shtml