High-Assurance SPIRAL

End-to-end Guarantees for Robot and Car Control

Franz Franchetti, Tze Meng Low, Stefan Mitsch, Juan Paolo Mendoza, Liangyan Gui, Amarin Phaosawasdi, David Padua, Soummya Kar, José Moura, Mike Franusich, André Platzer, Jeremy Johnson, Manuela Veloso

Cyber-physical systems (CPS) ranging from critical infrastructures such as power plants, to modern (semi) autonomous vehicles are systems that use software to control physical processes. CPS are made up of many different computational components. Each component runs its own piece of software that implements its control algorithms, based on its model of the environment. Every component then interacts with other components through the signals and values it sends out. Collectively, these components, and the code they run, drives the complex behaviors modern society have come to expect and rely on. Due to these intricate interactions between components, managing the hundreds to millions lines of software to ensure that the system, as a whole, performs as desired can be often unwielding.

In this article, an approach towards taming part of the complexity is described. The approach utilizes intrinsic multi-modal redundancies to detect brewing problems, provides formal guarantees for control algorithms, and automates the software production to implement these algorithmic ideas with guaranteed correctness.

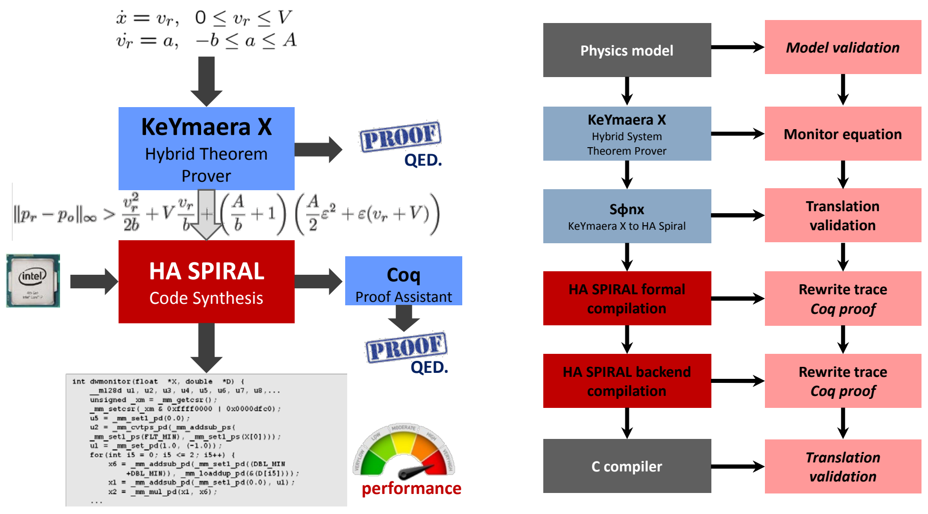
Specifically, the approach addresses the problem from three directions: 1) The desired behavior of the system and assumptions about the environment is described formally (in differential dynamic logic), and proven, over all valid executions, to perform correctly based on the formalized assumptions. 2) High performance monitor software that checks that the environment comply with the specified assumptions, and a proof that its implementation is a faithful representation of the mathematical specifications, are automatically synthesized from the differential dynamic logic model to reduce and even eliminate human coding error. 3) Side channel information such statistical noises are fused with traditional sensor inputs such as Global Positioning System (GPS), based on fundamental analytical redundancy, so as to establish that the inputs to the system (i.e. sensor readings) do not contradict the known physics of the system.

This approach has been demonstrated on both a remote controlled unmanned research ground robot, called the *Landshark*, and on an *American-built car*. In these demonstrations, the combination of formal methods for hybrid systems, automatic and provably correct code generation, and side-channel redundancy has been shown to detect and defend against *GPS spoofing* (manipulating of the GPS signal to make the victim believe to be at a different position), while protecting the car and robot from being driven into known obstacles. More importantly, these concepts are applicable in the CPS arena beyond unmanned ground vehicles or modern cars. Other domains for which the approach have shown applicability includes system components like pumps in power plants, and control of unmanned aerial vehicles.

Overview

Two formal systems are combined to provide end-to-end correctness guarantees from the control algorithm/physical model level down to the deployed implementation of the control algorithm. One or both systems formalize control approaches or self-consistency checks to ensure system safety. In either case, engineers then synthesize the final deployable software from a high level specification, and have guarantees that the synthesized software is correct and efficient.

The top level system is *KeYmaera X* [1], a theorem prover for differential dynamic logic [2]. With KeYmaera X it is possible to prove that a family of controllers, applied to a cyber-physical system with a given physical model will behave in a certain way. An example of a safety property that KeYmaera X can prove is that a robot with a controller based on the dynamic window control approach [3] will not hit an obstacle or another robot [4] (which is called *passive safety*). The


$\dot{x} = v_r, \quad 0 \le v_r \le V$
$\dot{v_r} = a, \quad -b \le a \le A$
KeYmaera X
Hybrid Theorem Prover
PROOF
QED.
$\|p_r - p_o\|_\infty > \frac{v_r^2}{2b} + V\frac{v_r}{b} + \left(\frac{A}{b}+1\right)\left(\frac{A}{2}\varepsilon^2 + \varepsilon(v_r+V)\right)$
HA SPIRAL
Code Synthesis
Coq
Proof Assistant
PROOF
QED.
performance
Physics model
Model validation
KeYmaera X
Hybrid System Theorem Prover
Monitor equation
Sϕnx
KeYmaera X to HA Spiral
Translation validation
HA SPIRAL formal compilation
Rewrite trace Coq proof
HA SPIRAL backend compilation
Rewrite trace Coq proof
C compiler
Translation validation

```
int dwmonitor(float  *X, double  *D) {
    __m128d u1, u2, u3, u4, u5, u6, u7, u8,...
    unsigned _xm = _mm_getcsr();
    _mm_setcsr(_xm & 0xffff0000 | 0x0000dfc0);
    u5 = _mm_set1_pd(0.0);
    u2 = _mm_cvtps_pd(_mm_addsub_ps(
    _mm_set1_ps(FLT_MIN), _mm_set1_ps(X[0])));
    u1 = _mm_set_pd(1.0, (-1.0));
    for(int i5 = 0; i5 <= 2; i5++) {
        x6 = _mm_addsub_pd(_mm_set1_pd((DBL_MIN
            +DBL_MIN)), _mm_loaddup_pd(&(D[i5])));
        x1 = _mm_addsub_pd(_mm_set1_pd(0.0), u1);
        x2 = _mm_mul_pd(x1, x6);
    ...
```

Figure 1: Overview of our end-to-end system (left) and the end-to-end proof argument (right).

dynamic window approach is suitable for robots driving circular trajectories. It computes admissible velocities that avoid collisions with obstacles, and from those it chooses a velocity that can be realized by the robot within a short time frame (the dynamic window) and bring it closer to some goal.

After verification, using a technique called *ModelPlex* [5], KeYmaera X synthesizes a provably correct mathematical condition (a *monitor*) with the following property. This generated monitor checks, at runtime, that the observed environment fits the verified model environment. When the observed behavior fits to the verified model as validated by the monitor execution at runtime, then the system execution is safe. However, when the monitor is violated, the system may have evolved beyond the model assumptions, which means that the system is potentially unsafe and will enter failsafe mode. This is similar to Simplex monitors [6] detecting when to switch between controllers.

Having derived a monitor condition that informs when the observed environment no longer fits to the assumed model, the remaining problem is to translate this monitoring expression into an efficient piece of software that performs the check at runtime. The SPIRAL system [7, 8, 9] is used to synthesize a software implementation from the monitoring expression. The core of SPIRAL is a rewriting system that manipulates SPIRAL's HCOL language (hybrid control operator language) into an equivalent expression that can be translated into code. Key requirements are that every HCOL expression has a mathematical interpretation, and each transformation performed on the HCOL expression must return a mathematically equivalent HCOL expression. The requirements, together, guarantee that the final code (when executed over the real numbers) would be a mathematically equivalent expression to the monitoring expression. Next, SPIRAL uses *interval arithmetic* [10] to implement this final code using floating point numbers available on current architectures. SPIRAL utilizes performance enhancing computer architecture features like SIMD vector instructions as well as aggressive compiler techniques (all of which are cast as mathematical rewrite rules) to produce highly efficient code.

Another component of the approach is to utilize *statistical and analytical redundancy* between multiple sensors that measure different quantities to establish that the current state of the system as understood by the controllers is *self consistent*, i.e., there is no intrinsic inconsistency in the measurements given the accuracy of the sensors. Statistical tests and analytical redundancy establish that location estimated through GPS and through a wheel encoder do not disagree more than the intrinsic inaccuracy of the respective sensors. This makes it possible to detect *GPS* *spoofing.* Other analytical redundancy methods for protecting againt compromised sensors include the estimation of vehicle speed from multiple sound channels obtained with microphones placed strategically on the car [11], estimation of vehicle altitude from correlating barometric pressure with GPS [12], and determining the posture of a robot via its camera view. All these methods protect against compromised sensors since they correlate measurements that have a complicated analytical relationship that cannot be easily maintained by an attacker.

This approach has been demonstrated on a number of manned and unmanned ground and air vehicles. The dynamic window monitor was deployed as an end-to-end example produced by KeYmaera X and SPIRAL. It was used both on the Landshark and the American-built car to prevent a malicious operator from crashing the vehicle into an obstacle. In addition, resilience to GPS spoofing while the monitor was running was demonstrated, by utilizing statistical and set based inconsistency detectors. Further the detection of replay attacks was demonstrated using a statistical test. In all these demonstration the critical code pieces were synthesized with the SPIRAL system. In addition very accurate speed estimation solely from vehicle sound was demonstrated. Finally, a quadcopter height controller with correctness guarantees was synthesized.

Proving Controllers Correct–And Catching Them If Not

Due to their impact on the real world, cyber-physical systems need to be safe. That poses a nontrivial but important challenge because it is not easy to get the control decisions exactly right to maintain safety of the physical system and its response through actuation, especially in light of the interaction with other agents in the environment. Formal verification has been identified as a powerful analysis technique to establish correctness guarantees about the behavior of the design or find issues as early as possible in the design process [13].

The development begins with a model of the system dynamics as a *hybrid system* [14, 15, 16, 2], which are mathematical models that feature both discrete and continuous dynamics. This flexible combination of dynamics is important for understanding systems with computerized or embedded controllers for physical systems since the latter are usually modeled continuously while the former are discrete. The development also begins with a precise formal definition of the safety property to be guaranteed.

The approach uses *differential dynamic logic* ($\mathsf{d}\mathcal{L}$ [17, 18, 19, 2, 20, 21]) as the language in which both hybrid systems model and desired correctness properties can be specified unambiguously. Differential dynamic logic also provides the systematic way of proving that the hybrid system satisfies such correctness properties and is implemented in the theorem prover KeYmaera X [1]. Once the hybrid systems model is proved to satisfy its desired correctness properties in KeYmaera X, the ModelPlex tactic [5], which is implemented in KeYmaera X, synthesizes provably correct monitor conditions that check compliance of the system with the verified model so that safety transfers to the real system implementation.

Model. To illustrate the principles in action, consider a ground robot that has to avoid collision with obstacles [4]. Let us consider a simple setting where the robot drives on a flat, even surface. It is equipped with a distance measurement sensor, such as radar or Lidar, so that the robot is able to detect drivable regions. Everything else is considered an obstacle (for example, walls or other robots), meaning that the robot is able to measure the distance to obstacles. The robot does so periodically according to its sampling period (for example, every 20 ms) when it decides on steering, acceleration and braking according. The decisions on acceleration and steering are input into actuators, which turn these into physical motion that is followed until the robot controller runs the next time (for example, 20 ms later). During that time, the decisions cannot be changed. That way, the robot can stitch together its trajectory by following circular arcs of varying radius, as in the dynamic window approach. The robot can avoid collisions with obstacles by stopping or by choosing appropriate values for steering that let it drive around obstacles.

In principle, obstacles could do the same. However, the number of constraints put on how obstacles will move should be kept low, so that the model fits many different kinds of motion. Hence, the model assumes a maximum velocity and otherwise allows any kind of motion (for example, walls stay put, while moving obstacles could even make sudden orientation changes and jumps in speed).

The model in equations (1)–(5) describes the decisions of obstacles *obst*, the control choices of the robot *robot*, and the entailed physical behavior *dyn*. It models the dynamic window approach [3] for collision avoidance and is described in detail in [4].

The model is shown in Fig. 2. The modeling idiom $t := 0; dyn, t' = 1 \ \& \ t \leq \varepsilon$ used in (1) describes the sampling period of the controller: the clock t with constant slope $t' = 1$, together with the condition $t \leq \varepsilon$, ensures that at most time ε passes between controller runs.

The obstacle model *obst* is very liberal. It only guarantees that obstacles will not exceed a maximum speed V. Otherwise, any behavior is allowed by choosing velocity $v_o := *$ nondeterministically, which even includes sudden orientation changes and jumps in speed.

The robot has three control choices. First, if the condition *safe* is satisfied it can choose its acceleration a_r and a new curve described by the rotational velocity ω_r and the curve radius r_c. Of course, not all choices are admissible, so the control branch ends in a test that allows only accelerations in the physical acceleration limits $-b \leq a_r \leq A$ between maximum braking $-b$ and maximum acceleration A. The condition further ensures that the robot is not spinning $r_c > 0$ and that the curve preserves planar rigid body motion: the curve preserves the robot's longitudinal speed $\omega_r r_c = v_r$. Second, the robot can stay stopped $a_r := 0$ without spinning $w_r := 0$, if it is stopped already. Finally, the robot can choose to just hit the emergency brakes $a_r := -b$ on its current curve unconditionally at any time.

These control choices entail physical behavior as described in *dyn*: the robot's position changes according to its speed and orientation ($p_r' = v_r d_r$), with speed in turn determined by acceleration ($v_r = a_r$), while orientation follows along the chosen curve ($\omega_r' = \frac{a_r}{r_c}$ and $d_r' = -\omega_r d_r^{\perp}$). The obstacle's position is modeled in a similar fashion. Note that $v_r' = a_r$ may result in negative speeds $v_r < 0$ upon braking $a_r < 0$, so the condition $v_r \geq 0$ ensures that hitting the brakes does not make the robot drive backwards.

Safety Property. Next, a safety property is needed in order to analyze the model *dw* from (1) formally. Intuitively, with only stationary obstacles around, at all times, everything the robot ever does has to result in positions different from obstacle positions, as captured in $p_r \neq p_o$. In the presence of moving obstacles, however, this condition needs to be relaxed, since guarantees are only possible about the robot at hand, not about the behavior of obstacles, as elaborated in [4, 22]. Hence, the model will guarantee *passive safety* $v_r \neq 0 \rightarrow p_r \neq p_o$, which means that there will be no collisions while the robot is driving. So, if a collision occurs at all, it is because a moving obstacle ran into the robot. Or if all agents are safe, there will be no collisions.

Eq. (6) below defines the requirements on the robot in a d$\mathcal{L}$ formula of the form *initial* $\rightarrow$ [*model*] *safety*. This means that, when the system starts in any initial state meeting the conditions *initial*, then *all* runs of *model* will end up with the safety condition *safety* being satisfied.

$$\begin{aligned} v_r = 0 \land A \geq 0 \land b > 0 \land \varepsilon > 0 \land V \geq 0 \\ \rightarrow [dw]\,(v_r \neq 0 \rightarrow p_r \neq p_o) \end{aligned} \quad (6)$$

The d$\mathcal{L}$ formula in (6) defines the starting condition *initial* as follows: the robot is stopped initially $v_r = 0$, and not malfunctioning, which includes a proper engine $A \geq 0$, functional brakes $b > 0$, a sampling period $\varepsilon > 0$, and it assumes that obstacles will not exceed $V \geq 0$. When started under these conditions, all executions of the model *dw* (denoted by the box operator $[dw]$ in d$\mathcal{L}$) guarantee passive safety ($v_r \neq 0 \rightarrow p_r \neq p_o$). The logical formula (6) can be analyzed in the hybrid system theorem prover KeYmaera X.

Verification. KeYmaera X applies sound axioms and proof rules to decompose the formula (6) into easier formulas, until only conditions in first-order real arithmetic remain. These conditions are finally checked for validity with a decision procedure for real arithmetic (quantifier elimination, for example, through cylindrical algebraic decomposition [23, 24]), resulting in a proof of the initial logical formula. While the verification of cyber-physical systems is certainly challenging, as is their design, KeYmaera X and its predecessor KeYmaera [25] have already been

$$dw \equiv (\mathit{obst};\ \mathit{robot};\ t := 0; \{\mathit{dyn}, t' = 1 \ \&\ t \le \varepsilon\})^* \tag{1}$$

$$\mathit{obst} \equiv v_o := *;\ ?v_o \le V \tag{2}$$

$$\mathit{robot} \equiv \begin{cases} a_r := *;\ \omega_r := *;\ r_c := *;\ ?\,(-b \le a_r \le A \wedge r_c > 0 \wedge \omega_r r_c = v_r) & \text{if } \mathit{safe} \\ a_r := 0;\ \omega_r := 0 & \text{if } v_r = 0 \\ a_r := -b & \text{unconditionally} \end{cases} \tag{3}$$

$$\mathit{dyn} \equiv p_r' = v_r d_r,\ v_r' = a_r,\ \omega_r' = \frac{a_r}{r_c},\ d_r' = -\omega d_r^{\perp},\ p_o' = v_o \ \&\ v_r \ge 0 \tag{4}$$

$$\mathit{safe} \equiv \|p_r - p_o\|_\infty > \frac{v_r^2}{2b} + V\frac{v_r}{b} + \left(\frac{A}{b} + 1\right)\left(\frac{A}{2}\varepsilon + \varepsilon(v_r + V)\right) \tag{5}$$

Figure 2: The hybrid system model of the joint discrete and continuous behavior of the robot and the obstacle. Control decisions are modeled in *obst* and *robot*, the physical motion is captured using differential equations in *dyn*.

used successfully to verify cars [26, 27], aircraft [28, 29], trains [30], robots [4], and surgical robots [31], and to verify the usual control schemes such as PID [32, 30]. For a tutorial on modeling and proving safety with d$\mathcal{L}$, see [33].

ModelPlex. Formal verification makes strong guarantees about the system behavior *if* adequate models of the system can be obtained. In any CPS design process, models are essential; but any model necessarily deviates from the real world. Faults may cause the system to function improperly, sensors may deliver uncertain values, actuators may suffer from disturbance, or the model may have assumed simpler ideal-world dynamics for tractability reasons or made unrealistically strong assumptions about the behavior of other agents in the environment. As a consequence, the *verification results obtained about models of a CPS only apply to the actual CPS at runtime to the extent that the model adequately represents reality.* A high-assurance CPS must be aware of the limitations in its design and equipped with means to detect deviations between design and reality.

The proofs so far formally show that a model of the robot is safe. In other words, the modeled family of robot controllers provably guarantees passive safety. The remaining task is to *validate* whether the model is adequate, so that the safety proof of the model transfers to the actual system implementation [34, 35]. ModelPlex [5] is a method to *synthesize correct-by-construction monitors for CPS by theorem proving automatically*: ModelPlex is based on the sound axioms and proof rules of d$\mathcal{L}$ [20, 21] to synthesize provably correct monitors that validate compliance of system executions with a model. The difficult question answered by ModelPlex is *what exact conditions need to be monitored* at runtime to guarantee compliance with the models and thus safety. ModelPlex enables tradeoffs between analytic power and accuracy of models while retaining strong safety guarantees.

At runtime, the ModelPlex monitors check the system behavior for model compliance. If the observed system execution fits to the verified model, then this execution is safe according to the offline verification result about the model. If it does not fit, then the system is potentially unsafe because it evolves outside the verified model and no longer has an applicable safety proof, so that a verified fail-safe action from the model is initiated to avoid safety risks (cf. Simplex [6]).

Since failures may occur and software attacks may happen, actual evolution must be monitored: the acceleration chosen by the controller must fit to the current situation (for example, accelerate only when safe), the chosen curve must fit to the current orientation, and no unintended change to the robot's speed, position, orientation, or knowledge about the obstacles occurred. This means, any variable that is allowed to change in the model must be monitored. In the example here, these variables include the robot's position p_r, longitudinal speed v_r, rotational speed ω_r, acceleration a_r, orientation d_r, curve r_c, and obstacle position p_o.

ModelPlex uses that the system is sampled periodically: for each variable there will be two observed values, one from the previous sample time (for example, positions p_r) and one from the current sample time (for example, p_r^+). It is not important for ModelPlex that the values are apart by exactly the sampling period, but merely that there is an upper bound (ε). A ModelPlex monitor checks in a provably correct way whether the evolution observed in the difference of the sampled values can be explained by the model. The verified hybrid system models themselves are not helpful as fast executable models, because they involve nondeterminism and differential equations. Hence, provably correct monitor expressions in real arithmetic are synthesized from a hybrid system model using an offline proof in KeYmaera X. These expressions exhaustively capture the behavior of the hybrid system models, projected onto the pairwise comparisons of sampled values that are needed at runtime. The full process is described in detail in [5].

ModelPlex monitor. Here, let us focus on a controller monitor expression synthesized from the model in (1)–(4) above, which captures all possible decisions of the robot that are considered safe. A controller monitor [5] checks the decisions of an (unverified) controller implementation for being consistent with the discrete model. ModelPlex automatically obtains the discrete model from model (1)–(4) with the ordinary differential equation (ODE) being safely over-approximated by its evolution domain. The resulting condition *monitor* in Fig. 3 (7), which is synthesized by a proof, follows the structure of the model: it captures the assumptions on the obstacle mon_o, the evolution domain from dynamics mon_{dyn}, as well as the specification for each of the three controller branches (braking mon_b, staying stopped mon_s, or accelerating mon_a).

The obstacle monitor part mon_o, see (8), says that the measured obstacle velocity d_r^+ must not exceed the assumptions made in the model about the maximum velocity of obstacles. The dynamics monitor part mon_{dyn}, see (9), checks the evolution domain of the ODE and that the controller did reset its clock ($t^+ = 0$). The braking monitor mon_b, see (10) defines that in emergency braking the controller must only hit the brakes and not change anything else (acceleration $a_r^+ = -b$, while everything else is of the form $x^+ = x$ meaning that no change is expected).[1] When staying stopped mon_s, see (11), the current robot speed must be zero ($v_r = 0$), and the controller must choose no acceleration and no rotation ($a_r = 0$ and $\omega_r = 0$), while everything else is unchanged. Finally, the acceleration monitor mon_a, see (12)–(13), when the distance is safe the robot can choose any acceleration in the physical limits $-b \le a_r^+ \le A$, a new non-spinning steering $c_r^+ \ne 0$ that fits to the current speed $\omega_r^+ c_r^+ = v_r$; position, orientation, and speed must not be set by the controller (those follow from the acceleration and steering choice).

The formula *monitor* in (7) synthesized with this correct-by-construction proof approach is the basis for code synthesis, as elaborated next.

Generating Code From a Mathematical Specification

Given a provably correct monitor specification that guarantees the desired behavioral (e.g. safety) properties, it is important that the instantiation of the specification as code is faithfully implemented. This ensures that all proven behavioral properties are preserved during the implementation process. In addition the implementation must be conservative in the presence of floating point rounding errors. As many proofs provided by formal methods are reasoned over real numbers—as opposed to floating point numbers found on computer systems—this difference in number representations, if not handled appropriately, may cause undesirable deviations from the specified model.

The SPIRAL system [7, 8, 9] synthesizes a conservative and faithful implementation from the mathematical specification through the successive application of identity rewriting. Each rewriting step replaces the input expression with a mathematically equivalent but more detailed expression that is more aligned to code. By ensuring that mathematical equivalence is preserved after each rewriting step, the correctness of the final implementation is guaranteed. Fig. 4 shows the overall flow. A multi-stage rewriting

[1]Note that unchanged obstacle position $p_r^+ = p_r$ means that the robot should not waste time measuring the obstacle's position, since braking is safe in any case.

$$
\begin{aligned}
\mathit{monitor} &\equiv \mathit{mon}_o \wedge \mathit{mon}_{dyn} \wedge (\mathit{mon}_b \vee \mathit{mon}_s \vee \mathit{mon}_a) && (7)\\
\mathit{mon}_o &\equiv \|d_r^+\| \le V && (8)\\
\mathit{mon}_{dyn} &\equiv 0 \le \varepsilon \wedge v_r \ge 0 \wedge t^+ = 0 && (9)\\
\mathit{mon}_b &\equiv p_o^+ = p_o \wedge p_r^+ = p_r \wedge d_r^+ = d_r \wedge v_r^+ = v_r \wedge \omega_r^+ = \omega_r \wedge a_r^+ = -b \wedge c_r^+ = c_r && (10)\\
\mathit{mon}_s &\equiv v_r = 0 \wedge p_o^+ = p_o \wedge p_r^+ = p_r \wedge d_r^+ = d_r \wedge v_r^+ = v_r \wedge \omega_r^+ = 0 \wedge a_r^+ = 0 \wedge c_r^+ = c_r && (11)\\
\mathit{mon}_a &\equiv -b \le a_r^+ \le A \wedge c_r^+ \ne 0 \wedge \omega_r^+ c_r^+ = v_r \wedge p_r^+ = p_r \wedge d_r^+ = d_r \wedge v_r^+ = v_r && (12)\\
&\quad \wedge \|p_r - p_o^+\|_\infty > \frac{v_r^2}{2b} + V\frac{v_r}{b} + \left(\frac{A}{b}+1\right)\left(\frac{A}{2}\varepsilon^2 + \varepsilon(v_r+V)\right) && (13)
\end{aligned}
$$

Figure 3: Synthesized safety conditions. The generated monitor captures conditions on obstacles mon_o, on dynamics mon_{dyn}, and on the robot controller's decisions on braking mon_b, staying stopped mon_s, and accelerating mon_a. The monitor distinguishes two observed values per variable, separated by a controller run (for example, p_r denotes the position before running the controller, whereas p_r^+ denotes the position after running the controller).

system [36, 37] consisting of a *backtracking and expansion* stage and multiple *recursive descent* and *confluent term rewriting* stages transforms an initial specification into a final piece of code, as explained in the remainder of this section.

Problem specification. Mathematical specifications are specified using SPIRAL's *hybrid control operator language (HCOL)*. In HCOL, an operator is a mathematical function that maps one or more real vectors to a real vector. Real scalars are treated as vectors of dimension one, and higher dimensional objects such as matrices are linearized into vectors. The following discussion focuses on Eq. (13), which is part of the full safety condition summarized in Fig. 3. Eq. (13) is written as HCOL operator as

$$
\begin{aligned}
\mathrm{SafeDist}_{V,A,b,\varepsilon} :\ & \mathbb{R}\times\mathbb{R}^2\times\mathbb{R}^2 \to \mathbb{Z}_2;\\
& (v_r, p_r, p_o) \mapsto \big(p(v_r) < d_\infty(p_r, p_o)\big) \qquad (14)
\end{aligned}
$$

with $d_\infty(\vec{x},\vec{y}) = \|\vec{x}-\vec{y}\|_\infty$, $p(x) = a_2x^2 + a_1x + a_0$, $a_2 = \frac{1}{2b}$, $a_1 = \frac{V}{b} + \varepsilon\left(\frac{A}{b}+1\right)$, and $a_0 = \left(\frac{A}{b}+1\right)\left(\frac{A}{2}\varepsilon^2 + \varepsilon V\right)$. This is essentially the same expression as (13) but makes explicit all data types and free parameters and expresses the computation explicitly in terms of higher-level mathematical objects such as polynomials and norms.

Breakdown rules and basic operators. The first step for translating (14) into an equivalent high performance implementation is to derive a top level *breakdown rule* that explains (14) in terms of SPIRAL's library of known mathematical objects expressed in the HCOL language, summarized in Fig. 5. The rule expressing this transformation for (14) is derived by KeYmaera X as

$$
\begin{aligned}
&\mathrm{SafeDist}_{V,A,b,\varepsilon}(.,.,.)\\
&\qquad \to \big(P[a_0,a_1,a_2](.) < d_\infty^2(.,.)\big)(.,.,.). \qquad (15)
\end{aligned}
$$

It closely mirrors the mathematical expression of the specification (14) and thus the original monitoring equation (13). As required, it expresses the semantics of the safety distance in terms of the HCOL library shown in Fig. 5. This leverages the HCOL formalization of well known mathematical objects such as *infinity norm*, *Chebyshev distance*, *scalar product*, or *evaluation of a polynomial* that are part of SPIRAL's library of mathematical objects and identities.

The goal of the rewriting process is to break HCOL operator specifications like (14) into expressions of *basic HCOL operators* through repeated applications of rules. The list of basic HCOL operators that are admissible in a fully expanded expression is shown in Fig. 6. Further, operations like $\circ$ and $\times$ (operator composition and Cartesian product) are also allowed.

Consider the example of *vector addition*, expressed through the basic operator $\mathrm{Pointwise}_{n\times n,(a,b)\mapsto a+b}$. The Pointwise operator takes two parameters (shown as subscripts), where $n \times n$ are the dimensions of the two

$$\begin{aligned}
\text{Pointwise}_{n,f_i} &: \mathbb{R}^n \to \mathbb{R}^n;\ (x_i) \mapsto \big(f_0(x_0),\dots,f_{n-1}(x_{n-1})\big)\\
\text{Pointwise}_{n\times n,f_i} &: \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n;\ \big((x_i),(y_i)\big) \mapsto \big(f_0(x_0,y_0),\dots,f_{n-1}(x_{n-1},y_{n-1})\big)\\
\text{Reduction}_{n,f_i} &: \mathbb{R}^n \to \mathbb{R};\ (x_i)_i \mapsto f_{n-1}(x_{n-1}, f_{n-2}(x_{n-2}, f_{n-3}(\dots f_0(x_0,\text{id}())\dots)\\
\text{Induction}_{n,f_i} &: \mathbb{R} \to \mathbb{R}^{n+1};\ x \mapsto (f_n(x, f_{n-1}(\dots)\dots),\dots,f_2(x,f_1(x,\text{id})),f_1(x,\text{id}),\text{id}())
\end{aligned}$$

Figure 6: Basic HCOL operators that have mathematical semantics but also can be seen as functional language constructs.

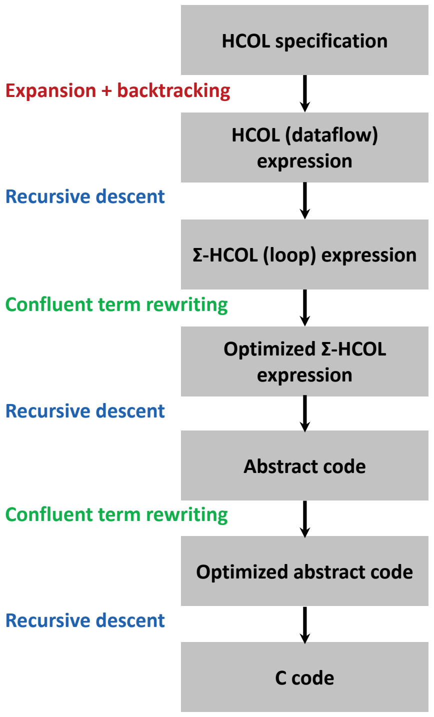


Figure 4: Code/proof co-synthesis as multi-stage rewriting system.

$$\begin{aligned}
\|.\|_\infty^n &: \mathbb{R}^n \to \mathbb{R};\ (x_i) \mapsto \max_{i=0,\dots,n-1} |x_i|\\
d_\infty^n(.,.) &: \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R};\ (x,y) \mapsto \|x-y\|_\infty^n\\
<.,.>_n &: \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R};\ \big((x_i),(y_i)\big) \mapsto \sum_{i=0}^{n-1} x_i y_i\\
(x^i)_n &: \mathbb{R} \to \mathbb{R}^{n+1};\ x \mapsto \big(x^0, x^1, \dots, x^n\big)\\
P[a_0,\dots,a_n] &: \mathbb{R} \to \mathbb{R};\ x \mapsto \sum_{i=0}^{n} a_i x^i
\end{aligned}$$

Figure 5: Library of mathematical objects expressed as HCOL operators.

input vectors, and $(a,b) \mapsto a+b$ is the mathematical operation that is performed on each pair of scalar elements from the two input vectors. Similarly, the Hadamard product (or element-wise multiplication) can be defined as $\text{Pointwise}_{n\times n,(a,b)\mapsto ab}$. More complicated operators can be defined through the composition of simpler operators through *HCOL operator expressions*. For example, the scalar (or dot) product can be described as

$$\begin{aligned}
<.,.>_n \to\ & \text{Reduction}_{n,(a,b)\mapsto a+b}\\
& \circ \text{Pointwise}_{n\times n,(a,b)\mapsto ab}\,. \qquad (16)
\end{aligned}$$

The recursive decomposition of higher level HCOL operators into basic operators is captured within the SPIRAL system as a *library of breakdown rules*. Fig. 7 collects all the breakdown rules needed to fully expand the safety distance monitor (14). By performing all the necessary substitutions as prescribed by the breakdown rules, the initial HCOL operator specification (14) is eventually translated into the finally expanded HCOL expression, shown in Fig. 8, consisting only of basic HCOL operators. This is the final result of the first stage in SPIRAL's rewriting system (backtracking and expansion, Fig. 4).

Code generation. The second stage in the code generation process is the translation of an HCOL expression into highly efficient C code. This is performed by a sequence of rewriting stages performing either a *recursive descend* or a *confluent term rewriting* phase. Logically, these

$$\begin{aligned}\mathrm{SafeDist}_{V,A,b,\varepsilon} &= \mathrm{Pointwise}_{(x,y)\mapsto x<y}\\ &\circ \Big(\big(\mathrm{Reduction}_{3,(x,y)\mapsto x+y} \circ \mathrm{Pointwise}_{3,x\mapsto a_i x} \circ \mathrm{Induction}_{3,(a,b)\mapsto ab,1} \big)\\ &\quad \times \big(\mathrm{Reduction}_{2,(x,y)\mapsto \max(|x|,|y|)} \circ \mathrm{Pointwise}_{2\times 2,(x,y)\mapsto x-y} \big) \Big)\end{aligned}$$

Figure 8: The final HCOL expression derived for the monitoring expression (13).

$$\begin{aligned}
&d^n_\infty(.,.) \to \|.\|^n_\infty \circ \mathrm{Pointwise}_{n\times n,(a,b)\mapsto a-b}\\
&\|.\|^n_\infty \to \mathrm{Reduction}_{n,(a,b)\mapsto \max(|a|,|b|)}\\
&< .,. >_n \to \mathrm{Reduction}_{n,(a,b)\mapsto a+b}\\
&\qquad \circ \mathrm{Pointwise}_{n\times n,(a,b)\mapsto ab}\\
&P[a_0,\dots,a_n] \to < (a_0,\dots,a_n), . > \circ (x^i)_n\\
&(x^i)_n \to \mathrm{Induction}_{n,(a,b)\mapsto ab,1}
\end{aligned}$$

Figure 7: Breakdown rules that express HCOL mathematical objects as basic HCOL objects.

steps are grouped into two stages using two separate sets of substitution rules.

In the first step, HCOL is translated into a lower level mathematical representation called Σ-OL, where loops are made explicit. For instance, Pointwise is translated into the following expression,

$$\mathrm{Pointwise}_{n\times n,f_i} \to \sum_{i=0}^{n-1} \mathrm{e}^n_i \circ \mathrm{Pointwise}_{1\times 1,f_i} \circ \big((\mathrm{e}^n_i)^\top \times (\mathrm{e}^n_i)^\top\big), \quad (17)$$

where e^n_i is a unit n-dimensional basis vector with the 1 at the ith position and $\times$ is the cross product. $(\mathrm{e}^n_i)^\top$ represents a gather operation and e^n_i represents a scatter operation. Similarly, the reduction operation is translated into Σ-OL by the rule

$$\mathrm{Reduction}_{n,(a,b)\mapsto a+b} \to \sum_{i=0}^{n-1} (\mathrm{e}^n_i)^\top.$$

At the Σ-OL level, optimizations performed by a traditional optimizing compilers are performed through substitution rules such as

$$\mathrm{Pointwise}_{n,f_i} \circ \mathrm{e}^j_n \to \mathrm{e}^j_n \circ \mathrm{Pointwise}_{1,f_j}\,. \quad (18)$$

```
# Hadamard Product
decl([i7], loopn(i7, n1,
  assign(nth(Y, i7),
    mul(nth(X, i7), nth(y1, i7)))))
```

```
# Reduction
decl([i4], chain(
  assign(nth(Y, V(0)), V(0)),
    loopn(i4, n1, decl([ s1 ], chain(
        assign(s1, nth(X, i4)),
        assign(nth(Y, V(0)), add(nth(Y, V(0)), s1))
)))))
```

```
# Scalar Product (optimized)
decl([i8], chain(
  assign(nth(Y, V(0)), V(0)),
    loopn(i8, n1, decl([ s2, s3 ], chain(
        assign(s3, nth(X, i8)),
        assign(s2, mul(s3, nth(y1, i8))),
        assign(nth(Y, V(0)), add(nth(Y, V(0)), s2))
)))))
```

Figure 9: Spiral's internal *icode* representation for the (top) Hadamard product, $\mathrm{Pointwise}_{n\times n,(a,b)\mapsto ab}$, (middle) Reduction, $\mathrm{Reduction}_{n,(a,b)\mapsto a+b}$, and (bottom) Scalar Product (after optimization). The *icode* is then pretty-printed in the desired programming language such as C. X is the input vector and Y is the output vector.

The above rule turns a program fragment that copies n pieces of data into contiguous memory addresses before applying the function f_i on each elements, into a program fragment that applies the same function on the appropriate piece of data, copies it into contiguous storage, and repeats for the remaining $n-1$ pieces of data. While functionally equivalent, the optimized program is more efficient since it parses through the data once.

Notice that the final Σ-OL expression is still a mathematical expression, but can be seen as highly optimized loop-based program that implements a mathematical function. In addition, because traditional compiler optimizations are implemented within SPIRAL as substitution rules, the correctness of the optimizations is ensured.

The second translation step translates a Σ-OL expression into an actual loop-based program, by means of a small set of *compilation rules* like

$$\begin{aligned}\text{Code}\big(y = (A \circ B)(x)\big) \rightarrow \big\{&\texttt{decl}(t),\\ &\text{Code}\big(t = B(x)\big), \text{Code}\big(y = A(t)\big)\big\}. \quad (19)\end{aligned}$$

Strong guarantees about loops, conditionals, and array accesses are inherited and deduced from the original expression. All this together guarantees that the program over the real numbers is a pure function that is mathematically equivalent to the original specification. Fig. 9 shows the final generated code for the Hadamard Product, Reduction operator, and scalar product over real arithmetic represented in SPIRAL's internal code representation, called *icode*. Rewriting Σ-HCOL to this internal code representation requires five translation rules, which are summarized in Table 1. By repeated application of these five rules, the icode representation for (13) is shown in Fig. 10.

Code optimization. SPIRAL generates verified code through a sequence of formally proved rewrite rules. The trace of the rewrite rules that were applied provides a certificate that a given program is correct. However, the generated code must be compiled. This last step should also be verified, and can be done through the use of a certified compiler such as CompCert[38]. As the goal is correct *and* efficient code, it is necessary to ensure the compiled code is optimized. While the performance of CompCert has improved, it usually do not yield code with performance that are comparable with those using state-of-the-art optimizing compilers such as Intel's C compiler. Nonetheless, it can be used since many of the optimizations a good compiler performs are accomplished through the transformations carried out during the rewrite process, such as the loop merging performed by (22).

Additional optimizations such as tiling, loop unrolling, and vectorization can be performed by source to source transformations and verified at the code level, and in many cases can be done at a higher mathematical level like the loop merging example. Even when the optimizations cannot be done at the mathematical level, the fact that the code is being generated allows various assumptions, like dependence, to be guaranteed which simplifies proofs of their correctness. This is illustrated by further optimizations applied to the scalar product example. After loop merging the generated code looks like

```
for (s=0, j = 0; j < 2*N; ++j) {
  s += x[j] * y[j];
}
```

This can be optimized by loop unrolling and vectorized code can be obtained by combining the operations in the unrolled loop.

```
s0 = 0;  s1 = 0; s = 0;
for (i = 0; i < M; ++i)
  for (j = 0; j < 2*N; j+=2) {
    s0 += x[j] * y[j];
    s1 += x[j+1] * y[j+1];
  }
s = s0 + s1;
```

The equivalence of the unrolled code and the initial code can be easily verified by induction. Alternatively, the vectorization can be derived and verified through higher level transformations; namely the rule

$$\langle .,. \rangle_{2n} \rightarrow \langle .,. \rangle_2 \circ \langle .,. \rangle_n \otimes \mathrm{I}_2 \qquad (20)$$

which uses the tensor product [39, 40] to obtain vectorized code.

These simple transformations can lead to a significant performance gain. Timings on an Intel Core i7-3770 processor running Ubuntu 14.04 with CompCert 2.5 show a speedup of 3.5 from just the unrolling. In order to benefit from vectorization it is necessary that CompCert be able to generate code with vector instructions; however, it is not required that CompCert perform vectorization as this can be done as shown. This shows that a certified compiler can be used, without sacrificing performance, when combined with source to source optimizations provided their is good support for basic compiler functionality such as register allocation and instruction scheduling.

Floating-point arithmetic. Finally, the difference between real and floating point number representation has to be tackled. A conservative approximation is attained through the use of *interval arithmetic* [10]. Each real number a is represented by an interval $[a_{\inf}, a_{\sup}]$ where the boundaries $a_{\inf}$ and $a_{\sup}$ are the floating point

$\text{Code}(y = (A \circ B)(x))$	$\rightarrow$ `decl(t, chain(` $\text{Code}(t = B(x))$, $\text{Code}(y = A(t))))$
$\text{Code}\left(y = \sum_{i=0}^{n-1} A_i(x)\right)$	$\rightarrow$ `chain(assign(`$y, 0$`),` `loop(`$i, [0..n-1]$, $\text{Code}(y += A_i(x))))$
$\text{Code}\left(y = (e_i^n)^\top(x)\right)$	$\rightarrow$ `assign(`$y, x[i]$`)`
$\text{Code}(y = e_i^n(x))$	$\rightarrow$ `assign(`$y[i], x$`)`
$\text{Code}(y = \text{Pointwise}_{1,f}(x))$	$\rightarrow$ `assign(`$y, f(x)$`)`

Table 1: Rewrite rules for translating Σ-HCOL to Spiral's *icode* representation.

numbers closest to a, such that $a_{\text{inf}} \leq a \leq a_{\text{sup}}$. This ensures that the actual (true) value is always bounded by a_{inf} and a_{sup}. Interval arithmetic then computes using the boundary values as opposed to the true value. For instance,

$$[a_{\text{inf}}, a_{\text{sup}}] + [b_{\text{inf}}, b_{\text{sup}}] = [\text{rounddown}(a_{\text{inf}} + b_{\text{inf}}), \text{roundup}(a_{\text{sup}} + b_{\text{sup}})].$$

Similarly, the multiplication of two intervals is given by

$$\begin{aligned}
&[a_{\text{inf}}, a_{\text{sup}}] \times [b_{\text{inf}}, b_{\text{sup}}] = \\
&\quad \big[\min\big(\text{rounddown}(-a_{\text{inf}} \times b_{\text{inf}}), \\
&\quad\quad \text{rounddown}(a_{\text{inf}} \times b_{\text{sup}}), \\
&\quad\quad \text{rounddown}(b_{\text{inf}} \times a_{\text{sup}}), \\
&\quad\quad \text{rounddown}(-a_{\text{sup}} \times b_{\text{sup}})\big), \\
&\quad \max\big(\text{roundup}(a_{\text{inf}} \times b_{\text{inf}}), \\
&\quad\quad \text{roundup}(-a_{\text{inf}} \times b_{\text{sup}}), \\
&\quad\quad \text{roundup}(-b_{\text{inf}} \times a_{\text{sup}}), \\
&\quad\quad \text{roundup}(a_{\text{sup}} \times b_{\text{sup}})\big)\big] \qquad (21)
\end{aligned}$$

By using proper floating point rounding modes, operations on the intervals guarantee that the result interval over floating point numbers includes the result that is over the real numbers. Implementing interval arithmetic efficiently on modern processors can be challenging. However, the implementation of interval arithmetics within SPIRAL leverages modern architecture features such as the single instruction multiple data (SIMD) vector instruction set to reduce the number of actual instructions executed by the processor.

Final monitor code. Introducing SPIRAL's interval arithmetics implementation to the icode

```
// icode implementation of Eq. (13) over the reals
func(TInt, "dwmonitor", [ X, D ],
   decl([i3, i5, q3, q4, s1, s4,
         s5, s6, s7, s8, w1, w2],
      chain(
         assign(s5, V(0.0)),
         assign(s8, nth(X, V(0))),
         assign(s7, V(1.0)),
         loop(i5, [0..2], chain(
               assign(s4, mul(s7, nth(D, i5))),
               assign(s5, add(s5, s4)),
               assign(s7, mul(s7, s8))
         )),
         assign(s1, V(0.0)),
         loop(i3, [0..1], chain(
               assign(q3, nth(X, add(i3, V(1)))),
               assign(q4, nth(X, add(V(3), i3))),
               assign(w1, sub(q3, q4)),
               assign(s6, cond(geq(w1, V(0)),
                        w1, neg(w1))),
               assign(s1, cond(geq(s1, s6),
                        s1, s6))
         )),
         assign(w2, geq(s1, s5)),
         creturn(w2)
)))
```

Figure 10: The implementation of the dynamic window monitor in SPIRAL's internal code representation in real arithmetic.

representation in Fig. 10 yields the C implementation in Fig. 11. It is implemented using the Intel C++ compiler's *intrinsic functions* to explicitly use the special vector instructions provided by the Intel SSE4 instruction set extension, and runs in approximately 100 processor cycles on an 3.6 GHz Intel Core i7 processor. Notice the complexity of the code. If manually implemented, the probability of an error being introduced would increase. However, this complexity is hidden from the programmer through the use of rewrite rules that are faithfully applied by SPIRAL. The faithful application of the rewrite rules ensure that the introduction of interval arithmetics preserve the input specifications (if compute with real numbers). In addition, it is also guaranteed that the real values is always bounded by the floating-point interval bounds, which ensures that the implementation is conservative.

Correctness proofs and guarantees. Since all transformations from specification to final code are rewrite rules that replace a mathematical object (expression) with another equivalent expression, the sequence of rule applications es-

```
// Final C/SSE 4.1 Implementation of Equation (13) for Intel Core i7 Processors
// This is a conservative high performance implementation using interval arithmetic
int dwmonitor(float  *X, double  *D) {
    __m128d u1, u2, u3, u4, u5, u6, u7, u8 , x1, x10, x13,
           x14, x17, x18, x19, x2, x3, x4, x6, x7, x8, x9;
    int w1;
    unsigned _xm = _mm_getcsr();
    _mm_setcsr(_xm & 0xffff0000 | 0x0000dfc0);
    u5 = _mm_set1_pd(0.0);
    u2 = _mm_cvtps_pd(_mm_addsub_ps(_mm_set1_ps(FLT_MIN), _mm_set1_ps(X[0])));
    u1 = _mm_set_pd(1.0, (-1.0));
    for(int i5 = 0; i5 <= 2; i5++) {
        x6 = _mm_addsub_pd(_mm_set1_pd((DBL_MIN + DBL_MIN)), _mm_loaddup_pd(&(D[i5])));
        x1 = _mm_addsub_pd(_mm_set1_pd(0.0), u1);
        x2 = _mm_mul_pd(x1, x6);
        x3 = _mm_mul_pd(_mm_shuffle_pd(x1, x1, _MM_SHUFFLE2(0, 1)), x6);
        x4 = _mm_sub_pd(_mm_set1_pd(0.0), _mm_min_pd(x3, x2));
        u3 = _mm_add_pd(_mm_max_pd(_mm_shuffle_pd(x4, x4, _MM_SHUFFLE2(0, 1)),
                               _mm_max_pd(x3, x2)), _mm_set1_pd(DBL_MIN));
        u5 = _mm_add_pd(u5, u3);
        x7 = _mm_addsub_pd(_mm_set1_pd(0.0), u1);
        x8 = _mm_mul_pd(x7, u2);
        x9 = _mm_mul_pd(_mm_shuffle_pd(x7, x7, _MM_SHUFFLE2(0, 1)), u2);
        x10 = _mm_sub_pd(_mm_set1_pd(0.0), _mm_min_pd(x9, x8));
        u1 = _mm_add_pd(_mm_max_pd(_mm_shuffle_pd(x10, x10, _MM_SHUFFLE2(0, 1)),
                               _mm_max_pd(x9, x8)), _mm_set1_pd(DBL_MIN));
    }
    u6 = _mm_set1_pd(0.0);
    for(int i3 = 0; i3 <= 1; i3++) {
        u8 = _mm_cvtps_pd(_mm_addsub_ps(_mm_set1_ps(FLT_MIN), _mm_set1_ps(X[(i3 + 1)])));
        u7 = _mm_cvtps_pd(_mm_addsub_ps(_mm_set1_ps(FLT_MIN), _mm_set1_ps(X[(3 + i3)])));
        x14 = _mm_add_pd(u8, _mm_shuffle_pd(u7, u7, _MM_SHUFFLE2(0, 1)));
        x13 = _mm_shuffle_pd(x14, x14, _MM_SHUFFLE2(0, 1));
        u4 = _mm_shuffle_pd(_mm_min_pd(x14, x13), _mm_max_pd(x14, x13), _MM_SHUFFLE2(1, 0));
        u6 = _mm_shuffle_pd(_mm_min_pd(u6, u4), _mm_max_pd(u6, u4), _MM_SHUFFLE2(1, 0));
    }
    x17 = _mm_addsub_pd(_mm_set1_pd(0.0), u6);
    x18 = _mm_addsub_pd(_mm_set1_pd(0.0), u5);
    x19 = _mm_cmpge_pd(x17, _mm_shuffle_pd(x18, x18, _MM_SHUFFLE2(0, 1)));
    w1 = (_mm_testc_si128(_mm_castpd_si128(x19), _mm_set_epi32(0xffffffff, 0xffffffff,
                                                        0xffffffff, 0xffffffff)) ???
         (_mm_testnzc_si128(_mm_castpd_si128(x19), _mm_set_epi32(0xffffffff, 0xffffffff,
                                                        0xffffffff, 0xffffffff))));
    __asm nop;
    if (_mm_getcsr() & 0x0d) {
        _mm_setcsr(_xm);
        return -1;
    }
    _mm_setcsr(_xm);
    return w1;
}
```

Figure 11: The implementation of the dynamic window monitor using interval arithmetic in Intel's SSE 4.1 instruction set. The shown monitor code runs in about 100 processor cycles on an 3.6 GHz Intel Core i7 processor.

tablishes mathematical equivalency of specification and final code. Over the real numbers the code is mathematically identical to the original specification. Over floating point numbers, the use of interval arithmetic in the resulting code ensures that the code is a conservative approximation. Numerical results are sound as the true answer is guaranteed to be in the result interval. Logical answers are sound as the answer is conservative: true/false/unknown. However, these guarantees are only true if the rules themselves have been implemented correctly.

Each rule that can be applied is formally verified so that the transformed expressions are guaranteed to be equivalent to the original expression. For example, the rewrite rule (16) is a special case of the more general rule

$$\begin{aligned}\text{Reduction}_{n,f} \circ \text{Pointwise}_{n\times n,g}&\\ \to \text{Reduction}_{n\times n,f\circ g},&\quad (22)\end{aligned}$$

which can be proven by induction on n. Alternatively, the validity of the special case in (16) can be verified, using the property that the scalar product is bilinear, and checking that the two sides agree on a basis. Note that reduction with plus is the linear transformation given by the $1\times n$ vector of ones, $(\mathbf{1}^n)^\top$, and the following computation shows that the left and right hand sides of (16), applied to an arbitrary pair of standard basis elements, are both equal to $\delta_{i,j}$, the Kronecker delta.

$$\begin{aligned}(\mathbf{1}^n)^\top(e_i^n \cdot e_j^n) &= (\mathbf{1}^n)^\top \delta_{i,j} e_i^n\\ &= \delta_{i,j}(\mathbf{1}^n)^\top e_i^n\\ &= \delta_{i,j} = \langle e_i^n, e_j^n\rangle\end{aligned}$$

Similarly Rule 17 can be verified by applying the left and right hand sides to an arbitrary pair of vectors (x, y) and checking that the j-th element of the results are the same.

$$\begin{aligned}&(e_j^n)^\top \circ \textstyle\sum_{i=0}^{n-1} \big(e_i^n \circ \text{Pointwise}_{1\times 1,f_i} \circ\\ &\qquad \big((e_i^n)^\top \times (e_i^n)^\top\big)\big)(x,y)\\ =\ & \textstyle\sum_{i=0}^{n-1} \big((e_j^n)^\top \circ e_i^n \circ \text{Pointwise}_{1\times 1,f_i} \circ\\ &\qquad \big((e_i^n)^\top \times (e_i^n)^\top\big)\big)(x,y)\\ =\ & \textstyle\sum_{i=0}^{n-1} \big(\delta_{i,j} \circ \text{Pointwise}_{1,f_i} \circ\\ &\qquad \big((e_i^n)^\top \times (e_i^n)^\top\big)\big)(x,y)\\ =\ & (e_j^n)^\top \circ \text{Pointwise}_{n\times n,f_i}(x,y)\end{aligned}$$

These calculations can be formalized and checked with a proof assistant such as Isabelle [41] or Coq [42]. Similar calculations allow us to verify the rule that merges the reduction and pointwise operators which optimizes the scalar product computation to use one instead of two loops.

When converting Σ-OL expressions to code we must verify that the resulting code correctly preserves the mathematical semantics of the expression. Once correctness is proven for the basic expressions such as reduction and pointwise, then an inductive proof can be obtained to prove that the code generated for arbitrary expressions built up form higher level operators such as composition and Cartesian product are correct. Similarly, optimizations that are traditionally performed by optimization compilers are formally written as rewrite rules in SPIRAL, thus proving that the optimizations applied by SPIRAL for performance reasons retain the correctness guarentees of the input specifications. For more details, see "Code Optimization as Rewrite Rules".

Anomaly Detection as Statistical Deviation from Nominal Behavior

This section presents a set of statistical methods for anomaly detection based on two observations: (a) Robot sensors usually produce data that is redundant but noisy, and (b) It is often feasible to specify a priori a model of nominal behavior for these redundancies, but not to fully specify all the anomalies that may occur. Thus, the resulting algorithms first build statistical models of nominal behavior, and then detect anomalies during execution by finding sequences of observations that do not fit the model of nominal behavior.

Nominal models from redundancy

Robots often produce redundant information about the world from various sources. This redundancy can occur at various levels, such as world state estimation, task completion time, or motion properties. This section explores the example of monitoring the robot's motion properties, since it is applicable to many mobile robots. Information about the robot's motion can be obtained from its wheel encoders, GPS sensors, IMUs, cameras, and the robot's input command, and localization algorithms that integrate these sensors, among others. Generally, given two simultaneous observations $\hat{\boldsymbol{x}}_t^1$ and $\hat{\boldsymbol{x}}_t^2$ obtained from different sources at time t, the algorithms assume that it is possible to map them to two comparable observations $\boldsymbol{x}_t^1 = f^1(\hat{\boldsymbol{x}}^1)$, and $\boldsymbol{x}_t^2 = f^2(\hat{\boldsymbol{x}}^2)$ that are expected to have similar values during nominal execution. For example, the robot's displacement between timesteps can be computed both from the robot's wheel encoder values, and from consecutive outputs of a sensor-fusing localization algorithm. Fig. 12a shows graphs of these two sources of information in the CoBot mobile robots [43] during nominal execution. The properties of the difference $\Delta\boldsymbol{x}_t = \boldsymbol{x}_t^1 - \boldsymbol{x}_t^2$ can be extracted from data of nominal execution. In particular, since many sensors have distributions that are approximately normal, the following examples will adhere to that distribution. Thus, the algorithm first creates a model $\boldsymbol{\theta}_0$ of nominal execution:

$$P(\Delta\boldsymbol{x}_t|\boldsymbol{\theta}_0) = \mathcal{N}(\mu, \sigma^2) \text{ where } \mu \in [\mu_-, \mu_+] \tag{23}$$

That is, the difference between the two sources is normally-distributed, with variance σ^2 extracted from nominal execution, and mean μ allowed to be within a small interval $[\mu_-, \mu_+]$ around 0. Other sensors and sources of information may have different distributions, but this section focuses on normal distributions as a useful example in robotics.

Statistical testing for anomalies

Given that the model $\boldsymbol{\theta}_0$ is given by a normal distribution, the detection algorithms use a Z-test to determine the probability of observing a set of observations at least as unlikely as X given nominal execution; this section describes the Z-test for one-dimensional observations, although extension to higher dimensions is straightforward.

Given the set of observations $X = \{\Delta\boldsymbol{x}_1, \Delta\boldsymbol{x}_2, \ldots, \Delta\boldsymbol{x}_n\}$, the algorithm estimates the probability that the true mean μ of the underlying distribution lies within $[\mu_-, \mu_+]$. That is, it calculates the probability $P(\mu_- \leq \mu \leq \mu_+)$. First, define the standardized sample mean Z:

$$Z(X) = \frac{\bar{X}(X) - \mu}{\sqrt{\sigma^2/|X|}} \quad \text{where } \bar{X}(X) = \frac{1}{n}\sum_{i=1}^{n} \Delta\boldsymbol{x}_i. \tag{24}$$

The standardized problem then becomes that of calculating $P(Z_- \leq Z \leq Z_+)$, where Z_- and Z_+ are calculated analogously to Z, replacing μ by μ_+ and μ_- respectively. Since these variables are in standard form, the desired proability is obtained using the standard cumulative normal distribution $\Phi(Z)$:

$$\begin{aligned} P(\mu_- \leq \mu \leq \mu_+) &= P(Z_- \leq Z \leq Z_+) \qquad (25)\\ &= P(Z \leq Z_+) - P(Z \leq Z_-)\\ &= \Phi(Z_+) - \Phi(Z_-) \end{aligned}$$

This probability is then compared to a threshold $P_{\min}$ to determine if the set X is too unlikely to come from $\boldsymbol{\theta}_0$.

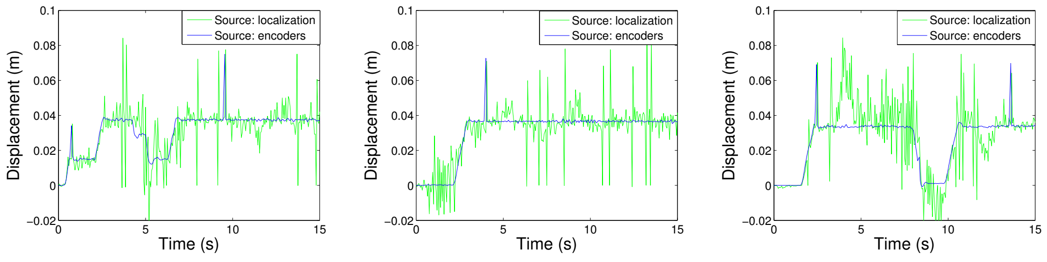
A Multi-Scale window approach to anomaly detection

Depending on the type of anomaly to be detected, different sets of observations may be analyzed for anomalies. This section focuses on analyzing *sequences* of observations to detect anomalies that start occurring at some time t_0, and affect the robot at any time $t \geq t_0$, such as those illustrated in Fig. 12; other work has analyzed sets of non-sequential but otherwise correlated observations [44].

During each time step t_k of execution, then, the algorithm searches for a time t_0 such that $P(\Delta\boldsymbol{x}_{t_0}, \Delta\boldsymbol{x}_{t_0+1}, \ldots, \Delta\boldsymbol{x}_{t_k}|\boldsymbol{\theta}_0)$ is too low to be considered nominal. One approach used in related work is to test every possible $t_0 \in [0, t_k]$ for anomalies. However, this approach grows linearly with the number of observations, which may be restrictive for online monitoring of long-deployment robots. Instead, the algorithm presented here uses an approach that tests windows of time of various scales to find anomalies. Thus, the detector creates N sets $X^0, X^1, \ldots, X^N$ of most recent observations on which to conduct a Z-test, where

$$X^i = \{\Delta\boldsymbol{x}_k, \Delta\boldsymbol{x}_{k-1}, \ldots, \Delta\boldsymbol{x}_{k-2^i}\}. \tag{26}$$

Then, the Z-test, previously discussed, is conducted on each of these windows of time.



(a) Nominal Execution (b) Subtle Anomaly (c) Clear Anomaly

Figure 12: Motion data gathered from the CoBot robots [43]. For the anomalies, the wheel encoders displayed varying levels of malfunctioning.

Algorithm 1 Multi-window approach to statistical anomaly detection.
Input: Sequence X of observations; Number of windows N; Nominal model $\boldsymbol{\theta}_0$
Output: true if an anomaly is detected, false otherwise.

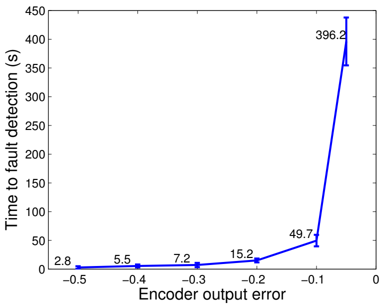
```
function DetectAnom($X = [\Delta\boldsymbol{x}_0, \Delta\boldsymbol{x}_1, \ldots, \Delta\boldsymbol{x}_k]$, $N$, $\boldsymbol{\theta}_0 = \{\sigma, \mu_-, \mu_+\}$)
    for $i \in \{0, 1, \ldots, N, \infty\}$ do
        $X^i \leftarrow \{\Delta\boldsymbol{x}_k, \Delta\boldsymbol{x}_{k-1}, \ldots, \Delta\boldsymbol{x}_{k-2^i}\}$    ▷ Extract data from window $i$
        $Z_+(X) = \frac{\bar{X}(X^i)-\mu_-}{\sqrt{\sigma^2/|X^i|}}$    ▷ Standardized deviations
        $Z_-(X) = \frac{\bar{X}(X^i)-\mu_+}{\sqrt{\sigma^2/|X^i|}}$
        $P \leftarrow \Phi(Z_+) - \Phi(Z_-)$    ▷ Probability that $\mu \in [\mu_-, \mu_+]$
        if $P < P_{\min}$ then
            return true    ▷ Probability too low, return failure
        end if
    end for
    return false    ▷ No probability found at any time scale
end function
```

Algorithm 1 summarizes the process of online statistical anomaly detection. The algorithm conducts the statistical Z-test on data coming from windows of N different sizes to find anomalies.

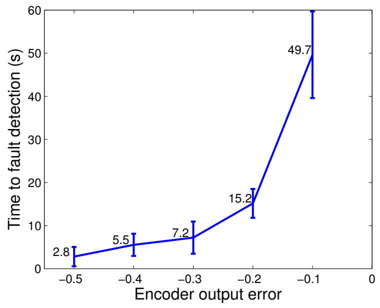
The time required to detect anomalies highly depends on the nature of the subtlety of the anomaly. Fig. 13 illustrates this: anomalies of different magnitudes were injected into one of the CoBot robot's wheel encoders: three of the wheel encoders work normally, but the fourth reports $(1-\epsilon)d$, where d is the displacement it would report if working normally. Thus, by varying ϵ form -0.5 to -0.1, the encoder reported half of its displacement, to 90% of its displacement. As ϵ approaches 0 –i.e., no anomaly–, the detection time asymptotically approaches infinity. Fig. 12 shows two anomalies: one with $\epsilon = 0.1$ and one with $\epsilon = 0.4$.

Detecting Sensor Inconsistencies and Secure State Estimation

This section focuses on malicious false-data-injection (FDI) attacks [45, 46, 47, 48] on the physical sensing resources in which an adversary potentially tampers (either remotely by hacking into the sensor software interfaces or by physically altering the sensing devices) the sensor data. Such attacks, if not detected promptly, might lead to inaccurate estimation of the vehicle state (such as its location and velocity) and trigger incorrect control actions with potentially devastating consequences. This section reviews a class of *model-based* approaches suited to the current application that use sensor data in conjunction with physics-based information (knowledge of vehicle kinematics models and nominal models of the sensors) to perform attack detection and secure state estimation. Model-based approaches, based on tight integration of system physics and sensor (data) characteristics, can be effective in terms of performance and implementability when sensor measurements can be linked to and represented in terms of physical state variables such as vehicle position and velocity. However, there might be other sensing modalities that may not be readily linked to the physical characteristics: information from



(a)



(b)

Figure 13: Time to fault detection as a function of the chosen fractional error ϵ. (a) shows all the experimental results, while (b) leaves out $\epsilon = -0.05$ for visualization of the remaining data. Error bars show one standard deviation.

these sensors might still contribute to the primary task of inconsistency detection, however, through purely sensor data driven processing. The interested reader may wish to refer to the side bar "Multi Modal Consistency" for additional details.

Overview. Model-based approaches are characterized by three crucial elements: *dynamical systems* (state-space) based representations of the vehicle kinematics, *sensor models* (both before and after potential FDI attacks), and the *inconsistency detection* and secure state estimation module. The remainder of this section discusses these three components in more detail and gives a theorem on detectable and undetectable attacks. The approach is based on linear models and provides an inconsistency detection procedure.

State-space models. A very simplistic abstraction of the vehicle kinematics may be obtained as

$$\mathbf{p}(t) = \mathbf{p}(0) + t\mathbf{v}(0) + \int_0^t \int_0^s \mathbf{a}(u)duds, \quad (27)$$

where $\mathbf{p}(t)$ and $\mathbf{v}(t)$ denote the position and velocity vectors respectively at time t (collectively the state $\mathbf{x}(t)$), and $t = 0$ corresponds to the origin of motion with $\mathbf{p}(0)$ and $\mathbf{v}(0)$ denoting the initial position and velocity respectively. The vector $\mathbf{a}(\cdot)$ corresponds to the instantaneous acceleration and, in control terminology, may be viewed as the input to the system. The acceleration may be assumed to be known up to a unknown but *bounded* (possibly disturbance) factor: in general, in this formulation it is assumed that at all times t, the deviation between the actual $\mathbf{a}(t)$ and its known (predictable) part $\mathbf{a}_{\text{known}}(t)$ is norm-bounded by a known constant $\bar{a}$. (In the worst case with no knowledge about the instantaneous acceleration, this constant corresponds to the vehicle's maximum possible acceleration in the given scenario.)

The important thing to note in the above is that, assuming the initial state $\mathbf{x}(0)$ at time $t = 0$ is known, the state uncertainty at any future time instant t may be captured by the relation

$$\mathbf{x}(t) \in \mathcal{C}_p^t(\bar{a}, \mathbf{x}(0)), \quad (28)$$

where $\mathcal{C}_p^t(\cdot)$ is a compact convex set depending on $\mathbf{x}(0)$ and $\bar{a}$ only. In other words, the vehicle kinematics provides (predictive) information about the system's state in terms of a bounded set of feasible states around the initial state; the associated prediction uncertainty is quantified by the size of $\mathcal{C}_p^t(\bar{a}, \mathbf{x}(0))$ which grows with t and $\bar{a}$.

Sensor models. In the nominal no-attack scenario, the n-th sensor, $n = 1, \ldots, N$, is assumed to measure a noisy linear function of the state at each sampling instant $k\Delta$. Here k, $k = 1, 2, \ldots,$ denotes the discrete sampling index and Δ the sampling period. Formally, the observation $\mathbf{y}_n(k\Delta)$ at the n-th sensor at $k\Delta$ is modeled as

$$\mathbf{y}_n(k\Delta) = H_n\mathbf{x}(k\Delta) + \mathbf{w}_n(\Delta), \quad (29)$$

where the matrix H_n specifies the sensing modality (such as GPS, wheel encoder, or IMU) and $\mathbf{w}_n(\Delta)$ the unknown sensing noise. The noise $\mathbf{w}_n(\cdot)$ is assumed to be norm-bounded but possibly state-dependent, i.e., there exists a continuous function $\bar{w}_n(\cdot)$ of the state such that

$\|\mathbf{w}_n(k\Delta)\| \leq \bar{w}_n(\mathbf{x}(k\Delta))$ for all k. Commonly used vehicle sensing resources which depend linearly on the instantaneous position and velocity may be cast in terms of (29), whereas, the bounded sensing noise is quite realistic for vehicular applications.

In the presence of FDI attacks, the sensor model (29) assumes the following form:

$$\mathbf{y}_n(k\Delta) = H_n\mathbf{x}(k\Delta) + \mathbf{w}_n(k\Delta) + \mathbf{b}_n(k\Delta), \quad (30)$$

where $\mathbf{b}_n(k\Delta)$ denotes the additional carefully crafted false data injected by the attacker into the nominal sensor measurements which is unknown to the system operator. Thus, from the system operator's viewpoint, both the sensor noise and the FDI attack contribute to the uncertainty of the measurement. The goal of the operator at any instant $K\Delta$ is to use the sensor data collected over all sensors at all times $k\Delta$, $k = 1, \ldots, K$ in conjunction with the knowledge of the vehicle kinematics to detect whether there has been an attack, i.e., $\mathbf{b}_n(k\Delta) \neq \mathbf{0}$ for some n and k, or not, and simultaneously obtain a *feasible* estimation of the vehicle state. This leads to inconsistency (attack) detector design discussed next.

Inconsistency detection and secure state estimation. In the following an *optimal* (to be discussed later) online recursive inconsistency detection and state estimation algorithm is presented. To this end, define for each n and k the set of feasible vehicle states $\mathcal{X}_n(\mathbf{y}_n(k\Delta))$ conforming to the measurement $\mathbf{y}_n(k\Delta)$, i.e.,

$$\begin{aligned}&\mathcal{X}_n(\mathbf{y}_n(k\Delta)) = \\ &\quad \{\mathbf{x} \;:\; \|\mathbf{y}_n(k\Delta) - H_n\mathbf{x}(k\Delta)\| \leq k_n(\mathbf{x}(k\Delta))\}\,.\end{aligned} \quad (31)$$

Now, consider the following recursive set membership filtering (RSMF) procedure, which generates recursively at each time instant $k\Delta$ a set-valued estimate $\mathcal{T}(k)$ of the vehicle's state $\mathbf{x}(k\Delta)$:

- *Initialization*: Set $\mathcal{T}(0) = \{\mathbf{x}(0)\}$.

- *Update*: At each $k \geq 0$, define the set

  $$\mathcal{T}_p(k+1) = \bigcup_{\widehat{\mathbf{x}} \in \mathcal{T}(k)} \mathcal{C}_p^1(\bar{a}, \widehat{\mathbf{x}}), \quad (32)$$

  where the set $\mathcal{C}_p^1(\cdot)$ corresponds to the set-valued one-step state prediction as a function of the acceleration-related norm-bound $\bar{a}$ and past state information $\mathcal{T}(k)$ as introduced in (28). Now, update $\mathcal{T}(k)$ as

  $$\mathcal{T}(k+1) = \underbrace{\mathcal{T}_p(k+1)}_{\substack{\text{one-step}\\ \text{prediction}}} \underbrace{\bigcap_{n=1}^{N} \mathcal{X}_n(\mathbf{y}_n((k+1)\Delta))}_{\text{innovation}}\,. \quad (33)$$

- *Detection, estimation and termination criteria*: If $\mathcal{T}(k+1) = \emptyset$ declare an attack and terminate; otherwise, declare $\mathcal{T}(k+1)$ to be the set of feasible vehicle states at time $k+1$ (in particular, any $\widehat{\mathbf{x}} \in \mathcal{T}(k+1)$ may be taken to be an estimate of $\mathbf{x}((k+1)\Delta))$ and continue the update step.

Note that, if in a given time horizon $[0, K\Delta]$, $\mathcal{T}(k) \neq \emptyset$ for all $k = 1, \ldots, K$, the test is inconclusive as to whether or not there has been no attack, i.e., $\mathbf{b}_n(k\Delta) = \mathbf{0}$ for all n, k: it might be possible that the attacker launched an *undetectable* attack trajectory $\{\mathbf{b}_n(k\Delta)\}$. In fact, undetectable attacks constitute non-zero attack trajectories $\{\mathbf{b}_n(k\Delta)\}$ that are carefully crafted such that they induce sensor observations that are feasible with respect to nominal or no-attack scenarios. The discussion on undetectable attacks will be revisited, but note, depending on the sensing model (the H_n matrices) and the noise characteristics, such attacks may exist. These undetectable attacks, when they exist, correspond to manipulating the sensor observations carefully (by the attacker) as a function of the geometry of the sensing models and the noise properties so as to induce tampered observations which nonetheless conform to all physical and sensing constraints. The following result presents important properties and optimality of the proposed RSMF algorithm (31)–(33).

Proposition 1 *The RSMF procedure outlined above satisfies the following properties:*

- *The procedure is* consistent*, i.e., if, in a given time horizon* $[0, K\Delta]$*, there is no FDI attack, then* $\mathcal{T}(k) \neq \emptyset$ *for all* $k = 1, \ldots, K$*. Further, in this case, the set* $\mathcal{T}(k)$ *exactly corresponds to the set of all feasible system states* $\widehat{\mathbf{x}}(k\Delta)$ *(including the true but unknown state* $\mathbf{x}(k\Delta)$*) that conform to the vehicle kinematics and (non-attacked) measurements in* $[0, K\Delta]$*.*

- *The procedure is* optimal *in the class of consistent attack detectors under similar knowledge constraints, i.e., in a given time horizon* $[0, K\Delta]$*, any non-zero attack sequence* $\{\mathbf{b}_n(k\Delta)\}_{n,k}$ *that is non-detectable by the RSMF procedure is also non-detectable by any other consistent attack detector under similar knowledge constraints.*

- *If the noise norm-bound functions* $k_n(\cdot)$*, see (29), are concave, the sets* $\mathcal{T}(k)$ *are convex for all* k*.*

- *If the collective observation matrix* $H = [H_1^\top \; H_2^\top \; \cdots \; H_N^\top]^\top$*, with* $\top$ *denoting matrix transpose, has full (row)-rank, the diameter of the set-valued estimation sets* $\mathcal{T}(k)$ *stay bounded, i.e., there exists a constant* $c > 0$ *such that*

$$\sup_k \sup_{\widehat{\mathbf{x}}, \hat{\widehat{\mathbf{x}}} \in \mathcal{T}(k)} \left\| \widehat{\mathbf{x}} - \hat{\widehat{\mathbf{x}}} \right\| \leq c. \tag{34}$$

Discussion. Implications of Proposition 1 are briefly described as follows. The consistency shows, in particular, the proposed detector has zero false alarm rate. The optimality in the class of all consistent detectors is clearly desirable. The convexity of the $\mathcal{T}(k)$ for all k (together with the fact that the sets stay bounded, see the final assertion of Proposition 1) implies that the detection-estimation step at each k (see (33)) reduces to a convex feasibility problem [49] and hence, may admit efficient numerical implementations such as by using the method of alternate projections. Finally, the (uniform) boundedness assertion implies that as long as the collective sensing model is sufficiently *informative* (essentially, an observability condition), the state estimation error (obtained by selecting an arbitrary member of $\mathcal{T}(k)$ as the estimate of $\mathbf{x}(k\Delta)$ at each instant $k\Delta$) under no-attack scenarios (respectively in scenarios involving detectable attacks) stays bounded at all times (respectively at all times till attack detection).

Undetectable attacks. Returning to the issue of attack undetectability, as noted earlier, the existence of undetectable attacks (and the set of all undetectable attacks) is, in general, jointly determined by the sensing models (the H_n matrices) and the noise characteristics. There is an important(sub)class of *fundamental* undetectable attacks are undetectable even in the limit of zero noise. These attacks are solely determined by the geometry of the sensing models. There is a rich literature on the characterization of such fundamental undetectable conditions for general linear time-invariant cyber-physical systems of the form studied in this paper [48, 50, 51, 52, 53, 54]. More recently, geometric control techniques have been employed to characterize FDI attack detection in cyber-physical systems in the presence of side information and more refined classification of attacks, for instance, characterizing attacks that can be sustained indefinitely without being detected and other related topics such as quickest detection of attacks (see [55]).

Tool Chain and Live Demos

The applicability of the approach discussed in this article was demonstrated on both the Landshark robot (shown in Fig. 14), a small scale commodity military robot, and an American-built car. In a series of demonstrations at the end of Phases I and II of the DARPA HACMS program, the three thrusts of the approach and their interdependence were displayed.

Emergency brake monitor. The KeYmaera X and SPIRAL systems were used to generate a monitor that ensures that the car/robot will not hit an obstacle between the current and subsequent execution of the monitor. In addition, if the assumed model of the environment no longer fits the observed environment, the monitor initiates an emergency stop. SPIRAL takes the monitoring expression derived by KeYmaera X as input and synthesizes a software implementation and the accompanying proof that together ensure whenever the software says the monitoring expression evaluates to false the true monitoring expression over the real numbers would have evaluated to false. Thus, the software implementation is proven to be conservative. This code is then deployed on the Landshark robot and the American-built car. Fig. 14 shows the moment when the KeYmarea-derived/SPIRAL-synthesized emergency monitor initiates an emergency stop of the Landshark robot to avoid hitting the obstacle.

This demonstration showed that a formal proof system, coupled with a provably correct method of generating conservative and efficient software implementation, can be used to generate quality

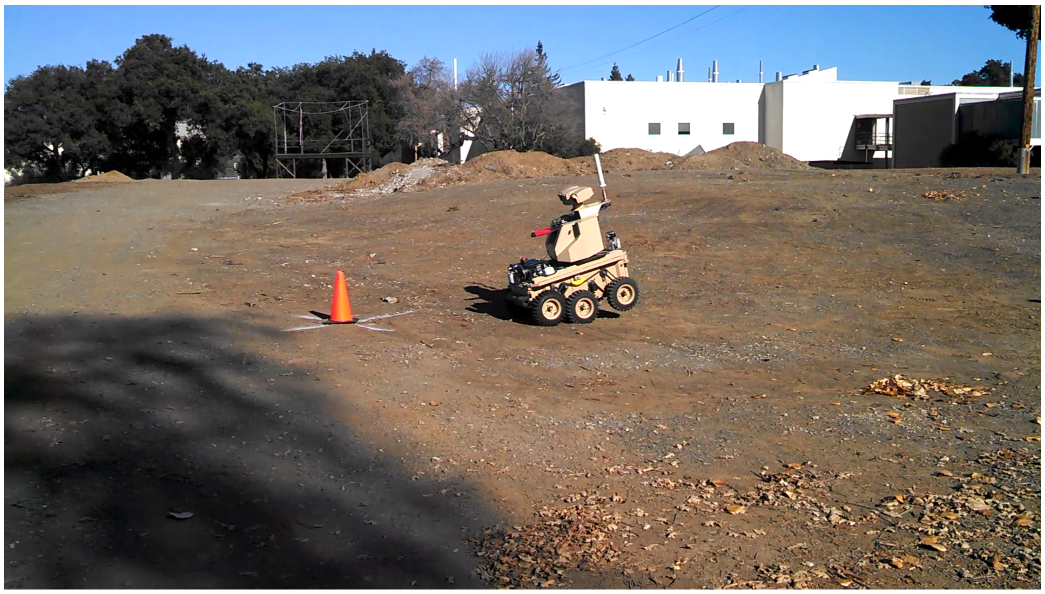


Figure 14: The Landshark robot stopping safely in front of an obstacle.

software that can be deployed on an actual production system. However, without ensuring that the inputs into the system are "reasonable" given the known operating environment, an adversary can still fool the monitor into performing outside of its operating assumptions by providing false/spoofed input signals. In the demonstration, the adversary was able to fool the monitor with false input signals (spoofed GPS that "teleported" the robot to a incorrect location), resulting in the Landshark running over the cone.

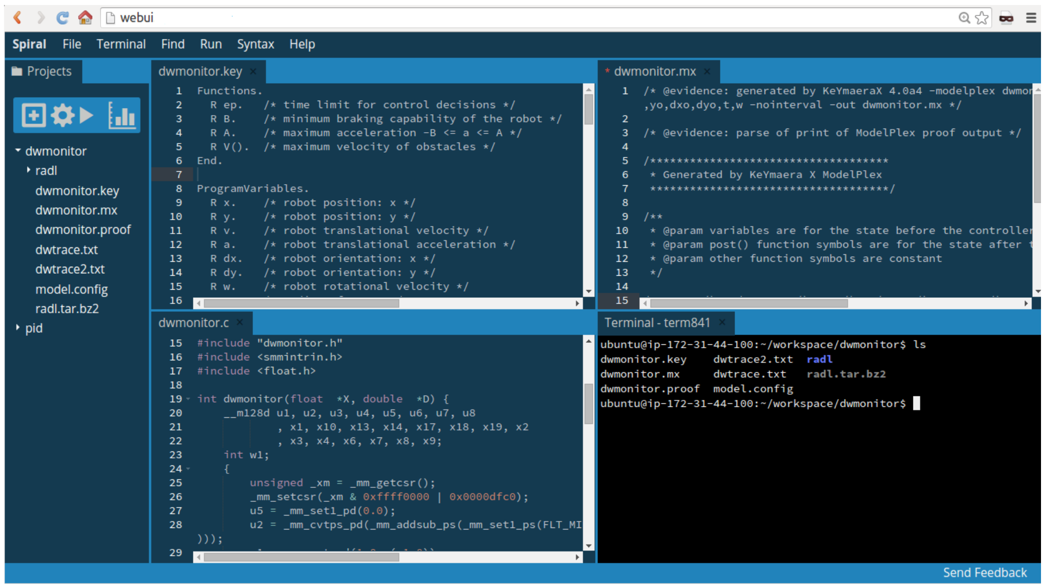
Defense against sensor spoofing. To address this issue, side channel redundancy was implemented to detect sensor spoofing. Specifically, inputs from the GPS and wheel encoders on the vehicle were fused statistically to detected when the mean of the difference between the two input signals deviated beyond a set threshold. These side channel redundancy algorithms were similarly generated by SPIRAL from their mathematical specifications. With the addition of side channel redundancy to the emergency brake monitor, changes of GPS values that were inconsistent with the inputs from the wheel encoders were detected. The presence of unreliable (i.e., spoofed) GPS inputs then triggered the emergency brakes, which stop the Landshark before the problem became catastrophic (i.e. the vehicle runs into an obstacle).

Tool chain. A cloud-hosted commercial grade tool chain with KeYmaera X and SPIRAL is accessible through a browser-based IDE (shown in Fig. 15). This makes the utilization of side channel redundancy, formal verification, and provably correct code generation accessible to a broader user base. Using the interface a user can perform a variety of tasks, such as studying and running examples, modifying existing projects, and building new projects, while the IDE provides levels of interaction ranging from click-and-run scripts to a command line window for expert users. Multiple users can log into the same instance for collaborative sessions, and users and projects are supported by standard scheduling and versioning tools in the cloud environment.

Along with exposing the full functionality of the core tools the interface has many of the general features typical of an IDE, such as context-sensitive menus, multiple tabs, online help, a text editor with language-specific syntax highlighting, and file downloads.

Conclusion

This article provides an overview of the *High Assurance SPIRAL* project, which is part of the DARPA HACMS program. The project brings together formal verification, code synthesis, and compilation aspects to provide end-to-end guar-



```
Functions.
  R ep.   /* time limit for control decisions */
  R B.    /* minimum braking capability of the robot */
  R A.    /* maximum acceleration -B <= a <= A */
  R V().  /* maximum velocity of obstacles */
End.

ProgramVariables.
  R x.    /* robot position: x */
  R y.    /* robot position: y */
  R v.    /* robot translational velocity */
  R a.    /* robot translational acceleration */
  R dx.   /* robot orientation: x */
  R dy.   /* robot orientation: y */
  R w.    /* robot rotational velocity */
```

```
/* @evidence: generated by KeYmaeraX 4.0a4 -modelplex dwmon
,yo,dxo,dyo,t,w -nointerval -out dwmonitor.mx */

/* @evidence: parse of print of ModelPlex proof output */

/*************************************
 * Generated by KeYmaera X ModelPlex
 *************************************/

/**
 * @param variables are for the state before the controller
 * @param post() function symbols are for the state after t
 * @param other function symbols are constant
 */
```

```
#include "dwmonitor.h"
#include <smmintrin.h>
#include <float.h>

int dwmonitor(float  *X, double  *D) {
    __m128d u1, u2, u3, u4, u5, u6, u7, u8
            , x1, x10, x13, x14, x17, x18, x19, x2
            , x3, x4, x6, x7, x8, x9;
    int w1;
    {
        unsigned _xm = _mm_getcsr();
        _mm_setcsr(_xm & 0xffff0000 | 0x0000dfc0);
        u5 = _mm_set1_pd(0.0);
        u2 = _mm_cvtps_pd(_mm_addsub_ps(_mm_set1_ps(FLT_MI
)));
```

```
ubuntu@ip-172-31-44-100:~/workspace/dwmonitor$ ls
dwmonitor.key    dwtrace2.txt  radl
dwmonitor.mx     dwtrace.txt   radl.tar.bz2
dwmonitor.proof  model.config
ubuntu@ip-172-31-44-100:~/workspace/dwmonitor$
```

Figure 15: The cloud computing interface to the integrated KeYmaera X and SPIRAL tool chain. The model and code generation of the dynamic window monitor is shown.

antees for control algorithms and safety monitors deployed on cyber-physical systems such as unmanned ground and air vehicles and state-of-the-art cars. In addition, the project leverages robotics and signal processing algorithms to detect attacks and establish trust in the available sensor readings. Together, the combined approach provides systematic and provable methods for designing controllers for specified desirable behaviors, generating implementations of the controllers with guarentees of correctness in the presence of floating point errors, and techniques and algorithms for detecting inconsistencies that may indicate the presence of an attacker.

This approach is orthogonal to, and builds upon traditional IT security defenses such as communication encryption and access controls. Most traditional security-in-defence techniques focuses on the securing the infrastructure and applications to ensure confidentiality, integrity and availability of the system. The presented approach provided added assurance in the form of guaranteed and provable behaviors, the absence of unintended errors in programming, and higher trust-worthiness of the sensor inputs.

This project also demonstrates that formal method techniques can be used to generate production quality code of significant complexity that can be deployed on, and used to operate actual cyber-physical systems. The feasibility and power of the presented approach was demonstrated at the final Phase I and Phase II demonstrations of the DARPA HACMS programs, where the team hardened the Landshark robot and an American built car to demonstrate the detection of GPS spoofing attacks and guaranteed passive safety. All implementations of algorithms discussed in this article were generated using an cloud-based tool front-end that integrated KeYmaera X and SPIRAL. This packaging of formal methods and side channel redundancy methods in a user friendly format shows a way forward to deploy these techniques on a larger scale for critical cyberphysical systems that require an extra high level of assurance and safety guarantees.

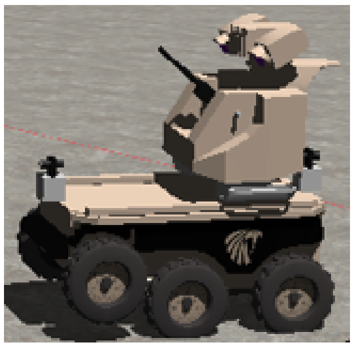


Figure 16: Landshark. The camera, the turret, and the paintball gun rotate.

Sidebar: Multi-Modal Consistency

A data-centric sensor fusion can be adopted to detect multi-modal sensor inconsistency, like inconsistency between a camera view and the orientation and posture of a robot. Based on the data received from the sensors, a model of the world is built and compared to the inputs from a different set of sensors. The model of the world and the inputs from the second of sensors must be consistent or an alarm would be triggered.

An example of this approach is demonstrated on the Landshark ground vehicle. Specifically, the Landshark is equipped with an auxiliary camera system that is used to detect inconsistencies in the values returned by the rotational sensors on the Landshark. It is important to note that, while the images captured by the camera (see details below) may not be readily linked to the vehicle physical kinematics as in the model-based approach discussed above, the image data can be compared with other invariants to detect inconsistencies.

The LandShark has four rotation degrees of freedom: 1) camera rotations around the horizontal and the vertical rotation axes; 2) turret rotations around the vertical axis; and 3) paintball gun rotations around the horizontal rotation. The LandShark has sensors to detect these rotation parameters. The key idea is to check the consistency between the data provided by the sensors and the images captured by the camera. At each time step, the rotation parameters returned by the sensors are used to generate cartoon images of what the camera should capture.

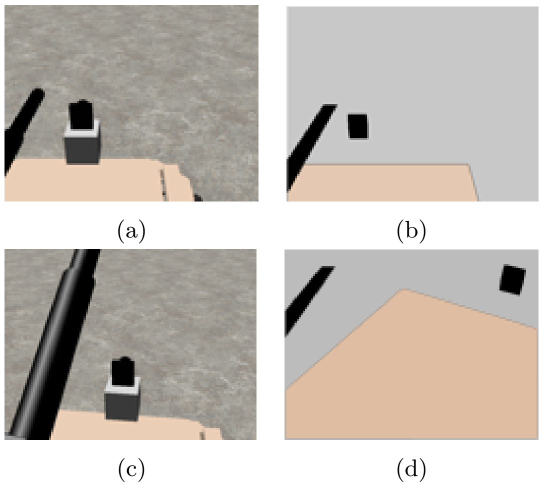


Figure 17: Two pairs of examples. Real image (17a) and cartoon image (17b) are consistent, showing that the sensors return the correct rotation parameters. Real image (17c) and cartoon image (17d) are inconsistent, showing that there is an e-attack.

The real images are captured by the camera in the same time step, and used as reference images. The cartoon images are subsequently compared with these real images to check for consistency. If they are inconsistent, the attack is flagged. If they are consistent, the sensors are assumed to be reliable.

Biograpy

Franz Franchetti is an Associate Research Professor at the Department of Electrical and Computer Engineering at Carnegie Mellon University. His research focusses on automatic performance tuning, program synthesis of mathematical kernels, and hardware/algorithm co-design. More information can be found at `http://www.ece.cmu.edu/~franzf/`.

Tze Meng Low is a Systems Scientist at the Department of Electrical and Computer Engineering at Carnegie Mellon University. His research focuses on high performance (hardware/software) implementations using analytical techniques and formal methods.

Stefan Mitsch is a Postdoctoral Researcher at the Department of Cooperative Information Systems at Johannes Kepler University and associated with the Logical Systems Lab of Carnegie Mellon University. His research focuses on modeling, formal verification, and runtime monitoring of cyber-physical systems.

Juan Pablo Mendoza is a PhD candidate in the Robotics Institute at Carnegie Mellon University. His research focuses on improving the robustness of autonomous robots at execution time via execution monitoring, anomaly detection and online learning. See www.cs.cmu.edu/ jmendoza/ for further information about his projects and publications.

Liangyan Gui is a PhD student in the Department of Electrical and Computer Engineering at Carnegie Mellon University. Her research focusses on image processing for cyber-physical systems.

Amarin Phaosawasdi is a PhD student in the Department of Computer Science at the University of Illinois at Urbana–Champaign. His research focus are program optimization and verification.

Jason Larkin is a Senior Research Engineer at SpiralGen, Inc. His research interests include modeling of complex phenomena, high performance and cloud computing, and open-source collaboration.

David Padua is a Donald Biggar Willett Professor of Engineering in the Department of Computer Science at the University of Illinois at Urbana-Champaign. His areas of interest include compilers and parallel computing. He is a Fellow of the IEEE and the ACM. More information can be found at `http://polaris.cs.uiuc.edu/~padua/`.

Soummya Kar is an Assistant Research Professor of ECE at Carnegie Mellon University. His research interests include performance analysis and inference in large-scale networked systems, multi-agent decision theory and stochastic systems. More information can be found at `http://www.ece.cmu.edu/~soummyak/`.

José M. F. Moura is the Philip L. and Marsha Dowd University Professor at Carnegie Mellon University, with the Electrical and Computer Engineering. Moura's research interests are in statistical signal and image processing. His research interests are in signal processing and data science. He is a Fellow from IEEE and AAAS, corresponding member of the Academia das Ciências of Portugal, and member of the US National Academy of Engineering. More information can be found at `http://www.ece.cmu.edu/~moura/`.

Mike Franusich is Vice President of Engineering at SpiralGen, Inc. in Pittsburgh. His work has focused on deploying emergent software technologies into vertical markets. More information can be found at `http://www.spiralgen.com/`.

André Platzer is an Associate Professor of Computer Science at Carnegie Mellon University. He develops the logical foundations of cyber-physical systems to characterize their fundamental principles and to answer the question how we can trust a computer to control physical processes. More information can be found at `http://symbolaris.com/andre.html`.

Jeremy Johnson is a Professor of Computer Science and Electrical and Computer Engineering at Drexel University. His research focuses on computer algebra, algebraic algorithms, program synthesis and verification, and high-performance computing and automated performance tuning. More information can be found at `https://www.cs.drexel.edu/~jjohnson/`.

Manuela M. Veloso is the Herbert A. Simon University Professor in the Computer Science Department at Carnegie Mellon University. She researches in Artificial Intelligence and Robotics, and is IEEE Fellow, AAAS Fellow, AAAI Fellow, and the past President of AAAI and RoboCup. See `www.cs.cmu.edu/~mmv` for further information, including publications.

References

[1] N. Fulton, S. Mitsch, J. Quesel, M. Völp, and A. Platzer, “Keymaera X: an axiomatic tactical theorem prover for hybrid systems,” in *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings* (A. P. Felty and A. Middeldorp, eds.), vol. 9195 of *LNCS*, pp. 527–538, Springer, 2015.

[2] A. Platzer, “Logics of dynamical systems,” in *LICS*, pp. 13–24, IEEE, 2012.

[3] D. Fox, W. Burgard, and S. Thrun, “The dynamic window approach to collision avoidance,” *IEEE Robot. Automat. Mag.*, vol. 4, no. 1, pp. 23–33, 1997.

[4] S. Mitsch, K. Ghorbal, and A. Platzer, “On provably safe obstacle avoidance for autonomous robotic ground vehicles,” in *Robotics: Science and Systems* (P. Newman, D. Fox, and D. Hsu, eds.), 2013.

[5] S. Mitsch and A. Platzer, “ModelPlex: Verified runtime validation of verified cyber-physical system models,” in *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings* (B. Bonakdarpour and S. A. Smolka, eds.), vol. 8734 of *LNCS*, pp. 199–214, Springer, 2014.

[6] D. Seto, B. Krogh, L. Sha, and A. Chutinan, “The Simplex architecture for safe online control system upgrades,” in *American Control Conference*, pp. 3504–3508, 1998.

[7] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, “SPIRAL: Code generation for DSP transforms,” *Proceedings of the IEEE, special issue on “Program Generation, Optimization, and Adaptation”*, vol. 93, no. 2, pp. 232– 275, 2005.

[8] F. Franchetti, M. Püschel, Y. Voronenko, S. Chellappa, and J. M. F. Moura, “Discrete Fourier transform on multicore,” *IEEE Signal Processing Magazine, special issue on “Signal Processing on Platforms with Multiple Cores”*, vol. 26, no. 6, pp. 90–102, 2009.

[9] M. Püschel, F. Franchetti, and Y. Voronenko, *Encyclopedia of Parallel Computing*, ch. Spiral. Springer, 2011.

[10] R. E. Moore, R. B. Kearfott, and M. J. Cloud, *Introduction to Interval Analysis*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2009.

[11] H. V. Koops and F. Franchetti, “An ensemble technique for estimating vehicle speed and geer position from acoustic data,” in *International Conference on Digital Signal Processing (DSP)*, 2015.

[12] V. Zaliva and F. Franchetti, “Barometric and GPS altitude sensor fusion,” in *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2014, Florence, Italy, May 4-9, 2014*, pp. 7525–7529, 2014.

[13] R. Alur, “Formal verification of hybrid systems,” in *EMSOFT* (S. Chakraborty, A. Jerraya, S. K. Baruah, and S. Fischmeister, eds.), pp. 273–278, ACM, 2011.

[14] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine, “The algorithmic analysis of hybrid systems,” *Theor. Comput. Sci.*, vol. 138, no. 1, pp. 3–34, 1995.

[15] T. A. Henzinger, “The theory of hybrid automata,” in *LICS*, pp. 278–292, IEEE Computer Society, 1996.

[16] J. M. Davoren and A. Nerode, “Logics for hybrid systems,” *IEEE*, vol. 88, no. 7, pp. 985–1010, 2000.

[17] A. Platzer, “Differential dynamic logic for verifying parametric hybrid systems.,” in *TABLEAUX* (N. Olivetti, ed.), vol. 4548 of *LNCS*, pp. 216–232, Springer, 2007.

[18] A. Platzer, “Differential dynamic logic for hybrid systems.,” *J. Autom. Reas.*, vol. 41, no. 2, pp. 143–189, 2008.

[19] A. Platzer, *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. Heidelberg: Springer, 2010.

[20] A. Platzer, "The complete proof theory of hybrid systems," in *LICS*, pp. 541–550, IEEE, 2012.

[21] A. Platzer, "A uniform substitution calculus for differential dynamic logic," in *CADE* (A. P. Felty and A. Middeldorp, eds.), vol. 9195 of *LNCS*, pp. 467–481, Springer, 2015.

[22] S. Mitsch, J.-D. Quesel, and A. Platzer, "From safety to guilty and from liveness to niceness," in *5th Workshop on Formal Methods for Robotics and Automation*, 2014.

[23] G. E. Collins, "Hauptvortrag: Quantifier elimination for real closed fields by cylindrical algebraic decomposition," in *Automata Theory and Formal Languages, 2nd GI Conference, Kaiserslautern, May 20-23, 1975* (H. Barkhage, ed.), vol. 33 of *Lecture Notes in Computer Science*, pp. 134–183, Springer, 1975.

[24] G. E. Collins and H. Hong, "Partial cylindrical algebraic decomposition for quantifier elimination," *J. Symb. Comput.*, vol. 12, no. 3, pp. 299–328, 1991.

[25] A. Platzer and J.-D. Quesel, "KeYmaera: A hybrid theorem prover for hybrid systems.," in *IJCAR* (A. Armando, P. Baumgartner, and G. Dowek, eds.), vol. 5195 of *LNCS*, pp. 171–178, Springer, 2008.

[26] S. M. Loos, A. Platzer, and L. Nistor, "Adaptive cruise control: Hybrid, distributed, and now formally verified," in *FM* (M. Butler and W. Schulte, eds.), vol. 6664 of *LNCS*, pp. 42–56, Springer, 2011.

[27] S. Mitsch, S. M. Loos, and A. Platzer, "Towards formal verification of freeway traffic control," in *ICCPS* (C. Lu, ed.), pp. 171–180, IEEE, 2012.

[28] A. Platzer and E. M. Clarke, "Formal verification of curved flight collision avoidance maneuvers: A case study," in *FM* (A. Cavalcanti and D. Dams, eds.), vol. 5850 of *LNCS*, pp. 547–562, Springer, 2009.

[29] J.-B. Jeannin, K. Ghorbal, Y. Kouskoulas, R. Gardner, A. Schmidt, and E. Z. A. Platzer, "A formally verified hybrid system for the next-generation airborne collision avoidance system," in *TACAS* (C. Baier and C. Tinelli, eds.), LNCS, Springer, 2015.

[30] A. Platzer and J.-D. Quesel, "European Train Control System: A case study in formal verification," in *ICFEM* (K. Breitman and A. Cavalcanti, eds.), vol. 5885 of *LNCS*, pp. 246–265, Springer, 2009.

[31] Y. Kouskoulas, D. W. Renshaw, A. Platzer, and P. Kazanzides, "Certifying the safe design of a virtual fixture control algorithm for a surgical robot," in *HSCC* (C. Belta and F. Ivancic, eds.), pp. 263–272, ACM, 2013.

[32] N. Aréchiga, S. M. Loos, A. Platzer, and B. H. Krogh, "Using theorem provers to guarantee closed-loop system properties," in *ACC* (D. Tilbury, ed.), pp. 3573–3580, 2012.

[33] J.-D. Quesel, S. Mitsch, S. Loos, N. Aréchiga, and A. Platzer, "How to model and prove hybrid systems with KeYmaera: A tutorial on safety," *STTT*, 2015.

[34] M. Leucker and C. Schallhart, "A brief account of runtime verification," *J. Log. Algebr. Program.*, vol. 78, no. 5, pp. 293–303, 2009.

[35] A. N. Srivastava and J. Schumann, "Software health management: a necessity for safety critical systems," *ISSE*, vol. 9, no. 4, pp. 219–233, 2013.

[36] N. Dershowitz and D. A. Plaisted, "Rewriting," in *Handbook of Automated Reasoning* (A. Robinson and A. Voronkov, eds.), vol. 1, ch. 9, pp. 535–610, Elsevier, 2001.

[37] J. W. Klop, "Handbook of logic in computer science (vol. 2)," ch. Term Rewriting Systems, pp. 1–116, 1992.

[38] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[39] J. R. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri, "A methodology for designing, modifying, and implementing Fourier transform algorithms on various architectures," *Circuits, Systems, and Signal Processing*, vol. 9, no. 4, pp. 449–500, 1990.

[40] F. Franchetti, F. de Mesmay, D. McFarlin, and M. Püschel, "Operator language: A program generation framework for fast kernels," in *IFIP Working Conference on Domain Specific Languages (DSL WC)*, vol. 5658 of *Lecture Notes in Computer Science*, pp. 385–410, Springer, 2009.

[41] T. Nipkow, M. Wenzel, and L. C. Paulson, *Isabelle/HOL: a proof assistant for higher-order logic*. Berlin, Heidelberg: Springer-Verlag, 2002.

[42] "The coq proof assistant reference manual," 2009.

[43] M. Veloso, J. Biswas, B. Coltin, S. Rosenthal, and R. Ventura, "Cobots: Collaborative robots servicing multi-floor buildings," in *International Conference on Intelligent Robots and Systems (IROS)*, October 2012.

[44] J. P. Mendoza, M. Veloso, and R. Simmons, "Focused optimization for online detection of anomalous regions," in *Proceedings of the International Conference on Robotics and Automation (ICRA)*, (Hong Kong, China), June 2014.

[45] A. A. Cárdenas, S. Amin, and S. Sastry, "Research challenges for the security of control systems," in *Proceedings of the 3rd Conference on Hot Topics in Security*, (San José, CA), pp. 1–6, July 2008.

[46] A. A. Cárdenas, S. Amin, Z. Lin, Y. H. and. C. Huang, and S. Sastry, "Attacks against process control systems: Risk assessment, detection, and response," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, (Hong Kong), pp. 355–366, Mar. 2011.

[47] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage, "Experimental security analysis of a modern automobile," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, (Oakland, CA), pp. 447–462, May 2010.

[48] A. Teixeira, D. Pérez, H. Sandberg, and K. H. Johansson, "Attack models and scenarios for networked control systems," in *Proceedings of the 1st ACM International Conference on High Confidence Networked Systems*, (Beijing, China), pp. 55–64, Apr. 2012.

[49] H. H. Bauschke and J. M. Borwein, "On projection algorithms for solving convex feasibility problems," *SIAM review*, vol. 38, no. 3, pp. 367–426, 1996.

[50] F. Pasqualetti, F. Dorfler, and F. Bullo, "Attack detection and identification in cyber-physical systems," *IEEE Transactions on Automatic Control*, vol. 58, pp. 2715–2729, Nov. 2013.

[51] Y. Mo and B. Sinopoli, "Integrity attacks on cyber-physical systems," in *Proceedings of the 1st ACM International Conference on High Confidence Networked Systems*, (Beijing, China), pp. 47–54, Apr. 2012.

[52] Y. Mo and B. Sinopoli, "False data injection attacks in control systems," in *Proceedings of the 1st Workshop on Secure Control Systems*, (Stockholm, Sweden), pp. 56–62, Apr. 2010.

[53] A. Teixeira, I. Shames, H. Sandberg, and K. H. Johansson, "Revealing stealthy attacks in control systems," in *Proceedings of the 50th Annual Allerton Conference*, (Monticello, IL), pp. 1806–1813, Oct. 2012.

[54] Y. Chen, S. Kar, and J. M. F. Moura, "Cyber-physical systems: Dynamic sensor attacks and strong observability," in *Proceedings of the 40th International Conference on Acoustics, Speech and Signal Processing*, (Brisbane, Australia), pp. 1752–1756, Apr. 2015.

[55] Y. Chen, S. Kar, and J. M. F. Moura, "Dynamic attack detection in cyber-physical systems with side initial state information."

IEEE Transactions on Automatic Control. Submitted. Initial Submission: Mar. 2015. Revised: Dec. 2015. [Online]: `http://arxiv.org/pdf/1503.07125v1.pdf`, Mar. 2015.

[56] "Black-i Robotics LandShark UGV."