Towards verified programming of embedded devices

Jean-Pierre Talpin Jean-Joseph Marty
Inria-Irisa
Rennes, France
{talpin, jean-joseph.marty}@inria.fr

Shravan Narayan Deian Stefan Rajesh Gupta
University of California at San Diego
La Jolla, CA, USA
{srn002, deian, gupta}@cs.ucsd.edu

***Abstract*—We propose a type-driven approach to building verified safe and correct IoT applications. Today's IoT applications are plagued with bugs that can cause physical damage. This is largely because developers account for physical constraints using ad-hoc techniques. Accounting for such constrains in a more principled fashion demands reasoning about the composition of all the software and hardware components of the application. Our proposed framework takes a step in this direction by (1) using refinement types to make make physical constraints explicit and (2) imposing an event-driven programing discipline to simplify the reasoning of system-wide properties to that of an event queue. In taking this approach, our framework makes it possible for developers to build verified IoT application by making it a type error for code to violate physical constraints.**

***Index Terms*—Formal verification, type and proof theory, program and model analysis, hybrid systems**

I. Introduction

Watches, appliances, cellphones, vehicles, grids, sensor networks, factories, supply chains: edge devices and systems of the Internet of Things (IoT) sense and actuate our living to bring it closer to the data-centers of an invading internet. Unfortunately, building IoT applications that are safe and secure is notoriously difficult. Though the popular media and some researchers have been quick to attribute fault to IoT developers being careless and not prioritizing safety and security [24], this fault placement is misguided. IoT developers are assigned with the heroic task of building applications that not only interact with the physical world, but are distributed and run on heterogeneous devices. Worse yet, the low profit-margins of IoT leave little room for serious testing and none for formal verification.

Indeed, most IoT developers take existing off-the-shelf components (e.g., Bluetooth stacks and LCD drivers), and glue them together using general-purpose programming languages like C. Unfortunately, these languages and the frameworks and libraries on top of them do not account for physical constraints. This makes it easy for developers to misuse hardware components—and either leave or introduce undefined behavior that can lead to physical damage.

Consider, for example, controlling a peripheral such as a stepper motor using Arduino's general-purpose IO (GPIO) interface.Writing to the GPIO pins directly to control the stepper is error-prone; languages like C make it easy for a developer to inadvertently change the direction of the motor, for example, by writing to the wrong pin number. Unfortunately, higher-level interfaces that encapsulate the underlying hardware protocol (e.g., Johnny Five's stepper API [21]) are also often unsafe. These high-level APIs allow developers to drive the motor *too fast*, which can cause the stepper motor to lose torque and behave unpredictably. This, in turn, makes it impossible for the software to keep track of the actual motor position—the very purpose of a stepper motor.

But, even if individual components are implemented correctly, *composing* them can introduce new classes of bugs and make it difficult to verify the system-level safety and security guarantees. Consider, for example, Arduino's stepper motor library which exposes a safe API that internally blocks to avoid running the motor too fast [5]. Though safe, this API does not compose well with other components. For example, an LCD display may need to be updated (e.g., to reflect the motor position) and a keyboard driver may need to handle user input (e.g., to stop the motor) concurrently, as the stepper motor is being driven. A blocking API is incompatible with this: we cannot, for instance, ensure that the display always reflects the state of the motor or that the motor can be stopped.

More generally, the guarantees of a component in isolation do not often translate to a larger system, when the component is composed with others. This is because components compete for resources such as memory and CPU time. For example, a stepper motor component implemented using Arduino's stepper library may operate correctly, i.e., it runs at a particular speed, in isolation, but in a larger system other blocking interfaces (e.g., the LCD driver) may break these guarantees and cause it to run slower. This may, in turn, violate the physical constraints of the hardware the motor itself is driving (e.g., the mount of a CNC milling machine).

In this paper, we propose to tackle the challenge of building correct IoT applications, and cyber-physical systems more broadly, by changing the programming model. In particular, we propose a type-driven approach that **(1)** makes physical and hardware constraints explicit **(2)** allows developers to specify and verify application-specific properties. By taking a type-driven approach we ensure that developers cannot write code that violates physical models and, for example, drive stepper motors too fast. At the same time, we can ensure that building large systems from smaller components preserves these guarantees and any properties specified by the developer (e.g., that the motor run at a particular speed). Our key insights, which we propose to implement in the F* programming language, are:

- We can encode physical constraints as *refinement types* on

978-3-9819263-2-3/DATE19/©2019 EDAA

parameterized hardware APIs. Refinement types allow us to ensure safety with respect to a model of the underlying hardware, and by using specialized real number solvers such as dReal, to match hardware models with the underlying physics.

- We can enforce system-wide properties (e.g., global invariants and component composition) by decomposing programs into a set of event-driven tasks. System-wide properties can then be expressed as temporal logic statements over events, and enforced using F*'s type system.

In the rest of the paper, we elaborate on these insights and describe our scalable, type-driven approach in detail (Section III). First, however, we describe the detail the underlying challenge of building an end-to-end correct cyber-physical system (Section II).

II. The difficulty of building IoT applications

To illustrate the challenges that arise when building IoT applications, we consider a simplified garage-door opener. Our garage-door opener is a simple CPS device that, on the press of a button, drives a stepper motor to open or close a garage door. We assume that motor needs eight clockwise rotations to open or close the door, in full.

To give developers fine-grained control of position, stepper-motors rotate in small increments. of this application. A stepper motor rotates 360° in four steps, i.e., 90° increments, by writing `HIGH` to each of the four control pins (and thus existing each of coil). To open the garage door, we can thus implement a function (`onOpenButtonPress`) that simply write to the four GPIO pins directly:

```
// Rotate the motor 8 times to open the door
void onOpenButtonPress() {
  for(rotation = 0; rotation < 8; rotation++)
    for(step = 0; step < 4; step++)
      activateCoil(step);
}
// Move motor a single step (activate a coil)
void activateCoil(int activePin) {
  for(pin = 0; pin < 4; pin++) {
    if (activePin == pin)
      digitalWrite(pin, HIGH);
    else
      digitalWrite(pin, LOW);
  }}
```

Physical constraints. Unfortunately, this code is not safe: it attempts to activate the different motor coils too fast. The step motor is a physical device that has constraints on the maximum rate at which it can step. At low speeds, the current applied to the GPIO pin has enough time to overcome the resistance and inductance of the motor's coils. However, at higher speeds, the charge build up in the coils is insufficient to generate the required torque for the motor. This results in the motor operating incorrectly, potentially leaving the door in a unknown state.

The above code is unsafe—Arduino's `digitalWrite` API does not enforce sufficient delays when operating the step motor. Instead, IoT developers (e.g., the Arduino stepper motor library [5]) account for this issue by inserting a `delay` after each step:

```
// Rotate the motor 8 times to open the door
void onOpenButtonPress() {
  for(rotation = 0; rotation < 8; rotation++)
    for(step = 0; step < 4; step++) {
      activateCoil(step);
      delay(...);
    }}
```

This blocking approach, unfortunately, may affect the responsiveness of the application. But, this is not fundamental—languages like JavaScript make it easy to do this in a non-blocking fashion [21].

Worse, delays are not enough: not all operational rates are safe. Step-motors have dead-zones and resonance frequencies that depend on their physical characteristics. Operating a step-motor in such unsafe zones results in imprecisions when taking a step. We explain safe-zones, dead-zones, resonance zones and the differential equations that model this behavior in the next section. Here we simply remark that the stepper motor must operate at a particular frequency corresponding to a *safe zone*—this physical constraint must be enforced when using the GPIO interface.

In practice, IoT developers account for this by using known safe rates (which they often find by trial and error). Unfortunately, even using high-level peripheral APIs that account for known-safe rates as such is not sufficient to guarantee safety. The external load on the motor may "shift" the safe zone from the "safe" default—to ensure safety, a verified system must model the physics and not rely solely on safe defaults.

Composition. To scale to large system, physical constraints must be enforced system-wide, in the presence of composition. For example, we must ensure that any sequence of door-open and -close operations operate respect the physical constraints of the motor. IoT developers rely on testing to capture such edge cases. But given the low-margins of IoT, these devices do not typically undergo exhaustive testing and therefore often miss such edge cases [9].

Even if we assume that we can handle all edge-cases, we run into further difficulties whenever we add new components. Consider, for example, extending our garage door openers with a wireless remote controller. A natural, but naive, approach to incorporating such a controller is to to listen for commands and open or close the door accordingly:

```
void onRemoteCommand(DoorCommand cmd) {
  if (cmd == OPEN) {
    onOpenButtonPress();
  } else if (cmd == CLOSED) {
    onCloseButtonPress();
  }}
```

This code—and the wireless driver and packet parsing code it relies on—shares resources with the code that operates the motor. Unfortunately, this also means that the code can reintroduce the safety bugs from before: the motor code may not get enough CPU time to run the motor at a safe rate. To guarantee the safety and correctness of our IoT device, we need to guarantee global properties of our system, i.e., we need to ensure that the composition of components, even when non-interfering, preserves the guarantees of each component.

Outlook. The simple example of a garage door opener demonstrates the challenges that IoT developers face when trying to build safe and correct applications: they must not only account for complex physical constraints when interfacing with a peripheral, but do so across the stack, in the presence of other software/hardware components. As we scale to larger cyber-physical systems, the need for programming languages and frameworks that make it easier to write safe and correct code is paramount. In the next section, we discuss a first step toward such a framework.

III. A Framework for Safe and Correct IoT

We propose to develop an event-driven framework for writing safe, correct, and performant IoT applications. Our framework builds on the F* programming language [40]. First, we leverage F*'s tools for writing verified safe programs, including its effect tracking, refinement types, and memory safety checks. Second, we leverage F*'s Kremlin backend [34] to emit efficient C code that can run on bare-metal microcontrollers such as the Arduino uno.

In the rest of this section, we outline the high-level design principles underlying this framework. In particular, we show how refinement types and our event-driven design paradigm can be used to write safe and correct IoT applications from the start. We also present some details on how we can model hardware to reflect physical constraints accurately.

Board Specifications. Our framework provides a minimalist Arduino library with safe primitives that respects specifications of the chosen microcontroller. Consider the `digitalWrite` API which writes a digital value to a particular GPIO pin. In the Arduino uno (amtel328p), pins 0 to 13 are analog pins, while pins 160 to 165 are digital pins. For safety, we must guarantee that we only write digital values to the appropriate pins. This guarantee, can be encoded using refinement types:

```
private type pin = p:UInt8 {
  (p>=0 && p<=13) \/ (p>=160 && p<=165)}
type digital_pin = p:pin {p >= 0 && p <= 13}
type analog_pin = p:pin {p >= 160 && p <= 165}
```

Refinement types combine data types with boolean predicates. The type `pin` in the above code is a byte type (`UInt8`) with a predicate that states its value must lie between 0 and 13 or between 160 and 165. (The range check is performed at compile time with the help of a SMT solver and does not incur any runtime performance penalty.) As shown in Fig. 1, this can be used to ensure that the `digitalWrite` API only accepts parameters of type `digitalPin`—and, in turn, guarantee that the IoT developer cannot misuse the board by, for example, accidentally using `digitalWrite` to write to an analog pin.

We note that the interface of our hardware platform is still preliminary. Nevertheless, in large CPS designs, such architecture models play a central role [28]. We hence seek structured, categorized, hierarchical representations of such interfaces, in the spirit of [3], [19], and propose to incorporate such interfaces using F*'s functors and modules. In particular, we seek models which allow static verification of program execution against the hardware interface. These models would incorporate logical, timed, concurrent abstractions of the hardware behavior.

```
module Arduino    .../...
(** write to a logical pin *)
assume val digitalWrite : pin:digital_pin ->
↪  value:digital_value ->  HST.ST (unit)
  (requires fun h -> True)
  (ensures fun h0 x h1 -> M.modifies (M.loc_none) h0
  ↪  h1)
(** read to a logical pin *)
assume val digitalRead : pin:digital_pin -> HST.ST
↪  (digital_value)
  (requires fun h -> True)
  (ensures fun h0 x h1 -> M.modifies (M.loc_none) h0
  ↪  h1)
```

Fig. 1. Type refinements in the Arduino platform interface

Specifying and Enforcing Constraints. Refinement types also allow us to specify higher-level constraints on code. For example, we can ensure that the stepper motor function `step` provided by our framework can only be called at a safe rate globally. Specifically, we enforces a delay between calls to the step motor with the help of two events. The `step` API, when called, creates a new globally visible event `Stepping`. Once a sufficient amount of time has passed, the framework automatically creates a globally visible `StepComplete` event. Consider the signature of the `step` API provided by our framework(We elide some of the syntax to simplify exposition).

```
let step : (s:stepper) -> (dir:step_direction) ->
↪  HST.ST (unit)
  (requires fun h -> not exists Stepping after
  ↪  StepComplete)
  (ensures fun h0 x h1 -> exists Stepping && future
  ↪  StepComplete)
```

The precondition of `step` (the `requires` clause above) enforces that the last event must be a `StepComplete` event, ensuring the stepper motor is not currently stepping. Calling `step` when this is not provable leads to a compile-time error. The postcondition of `step` (the `ensures` clause above) states that the `Stepping` event is generated by the `step` and that, at some point in the future, the `StepComplete` event is created. This allows us to guarantee any program that uses the `step` API cannot run the motor too fast, provided that the duration of step meets the expected ranges specified in the hardware and physical model interfaces (describe below). Note that the preconditions and postconditions of the `step` API can, in the software specification, be expressed as linear temporal logic (LTL) statements [32]. These LTL statements are specified as first order statements that apply to the sequence of all events generated by the system. To ensure no performance overhead at runtime, these LTL statements may be expressed as ghost computation—statements that exist at compile-time to help prove safety, but are not executed at run-time.

Our approach of enforcing LTL statements on the flow of events may be naturally used to prove system-wide properties. Consider the following statement:

```
not exists Stepping next Stepping
```

Applying this statement to the `main` function's post condition would provide a guarantee that the stepper motor never took a second step before the first step issues the `StepComplete`

event, i.e., there is always adequate delay between each step in the entire application. Indeed the same mechanism can be used to prove additional properties about the entire system. For example, in our garage-door opener, we can state that pressing the open button on the remote always results in the opening of the garage door:

```
exists (OpenButtonPress eventually GarageDoorOpen)
```

Composition. Our framework provides a single threaded co-operative scheduler that runs the IoT application's tasks. Each task runs uninterrupted, to completion. This ensures that the developer is not forced to reason about concurrency. To ensure multiple components' tasks may be modularly composed, our framework ensures that all tasks are finite in size. This ensures that no component can run a long or infinite task that uses all the CPU time. For example, we can ensure that the code that handles wireless remote commands cannot interfere with the stepper motor code and thus cause the motor to operate outside the safe zone.

Internally, we rely on refinement types and a *tick monad* to restrict the size of tasks to a fixed number of CPU operations. Attempting to run a task larger than the maximum allowed task size would raise a compile-time error. The maximum task size is specified by the IoT developer and is checked against the timing constraints exposed from the hardware interface and the physical model.

Modeling Peripherals. Our framework requires an accurate model of hardware along with specifications of safe values for hardware parameters (such as the maximum step rate of a stepper motor). To ensure safe operation of hardware, the framework specifies "safe zones"—values of hardware parameters that are safe along with proofs of their safety. The APIs exposed by our framework always enforce these models to ensure that IoT applications operate the hardware safely.

Consider the modeling of a stepper motor controller. The motor controller must certify the maximum step rate I, and precisely take it into account the timed semantics as a refinement type in the controller model. The first step in this process is to use existing models of stepper motors such as those available in Matlab (shown in Fig. 2). These models allow us to isolate the time interval below which a coil cannot charge up to 63% before phasing off. Next, we encode $I(t)$ in the refinement types of hardware APIs exposed by our framework. $I(t)$ is specified as a real valued ghost variable on the parameters of hardware APIs. This ghost variable generates real value constraints whose solution would indicate safety of the specified step rate. We submit the resulting constraints to dReal [23] as shown in Fig. 3, to verify that the equations are δ-decidable for the step frequency interval `F` of the chosen platform.

While the solution of Fig. 2 can be specified by simple SAT-modulo real constraints and solved using dReal, the proof that the motor actually stalls when the step frequency cross the boundary of I would require a more elaborate electro-mechanical model of the motor. Such models can be build in Matlab, dReal, HCSP [43], or differential dynamic logic

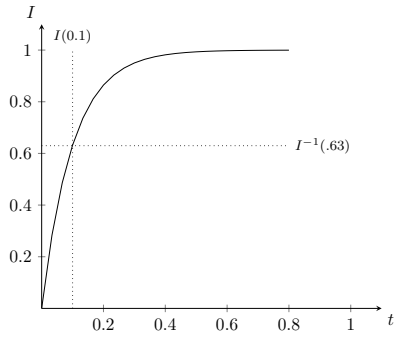

Fig. 2. A formalization of the induction $I(t) = (1 - e^{-Rt/L})V/R$ of a coil with resistance $R = 10$, voltage $V = 5$ and inductance $L = 2$. The 63% induction threshold suggested in [1] is reached at the maximal frequency of $10hz$ (solved using the LaTeX package `tikz`).

```
// requirements
#define minI 0.63
#define minF 0.01
#define maxF 10
// model constants
#define R 10 // resistance
#define V 5  // voltage
#define I 2  // inductance
// step frequency interval
[minF, maxF] F;
// induction function of 1/F
#define I (* (- 1 (exp (- 0 (/ R (* F L))))) (/ V
↪  R))
// invariants for I
invt: (V>=0); (F>0); (L>0); (R>0);
goal: I >= minI
```

Fig. 3. Compilation of type refinement constraints into a dReal problem

(ddL) [30] and the proof of their differential invariants using theorem provers like Isabelle and KeymaeraX [31].

To compose such models and their invariants as contracts, we rely on approaches such as those of [26], which give a method for composing models in ddL using differential invariants. Fig. 4 shows hows the approach of [26] is applied to the model of a controlled tank in an automated factory. Specifically, this approach can be used to mechanically prove invariants of individual components expressed by hybrid ddL specifications (e.g., the tank's analog water-level gauge and the tank's digital controller) and then automate the construction and proof of the combined invariants with respect to the composed system—the controlled tank.

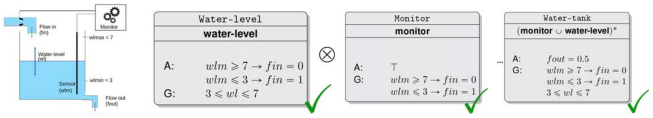

Fig. 4. Automated composition of contract in ddL.

Variability: So far, we have outlined a method for building verified IoT applications that takes into account formalized specifications of the hardware and physical environment. We outlined the some of the required artifacts, which consider a mathematically perfect model of time as temporal logic (software), real-time logic (hardware) and differential logic

(physics). This, unfortunately, is unsatisfactory: every IoT device (e.g., the Arduino) is subject to errors and, more generally, variability [41].

One expensive solution is to fix the underlying (Arduino) platform with additional hardware: a real-time clock or high-precision timers [18] which will be used by the software model (e.g., `delay` function in our exmpl). An alternative approach is to develop a stochastic model of variation between the built-in clock and a real-time clock to increase the accuracy of the clock. But clocks are not the only concern: hardware variability also entails the loss of precision of modern hardware due to increasingly small fabric. Nano-scale circuits become less reliable and more expensive to produce. They no longer behave like precisely chiseled machines with tight tolerances. Modern computing tends to ignore the variability in behavior of underlying system components from device to device, their wear and tear over time (MTBF), or the environment in which the computing system is placed. The NSF Variability Expedition (variability.org) envisions a computing world where system components, led by proactive software, routinely monitor, predict and adapt to the variability of manufactured systems.

IV. Related Work and Discussion

High-precision stepper motors are used in a myriad of applications: hard drives, computer-assisted design, precision mechanics, laser and optical equipment, medical equipment, etc. In high value applications such as medicine, the stepper motors used are, fortunately, subject to very precise specification requirements (e.g., the R and L of a step motors). Even so, the use of hardware such as stepper motors poses challenges with scale. Cyber-physical systems found in transportation, factories, grids, usually contains thousands of such individual control loops which result in intertwined software, hardware, mechanics and physics models. There have been several works that attempt to address this problem. These works approach the problem either in terms of "correctness by construction"—focusing on composition, modularity and end-to-end correctness, or in terms of "deep specification'—focusing on abstraction and accuracy of hardware modeling. We describe some of these below.

Correctness by Construction. Concepts of interfaces and contracts have been used to address compositionality in system design [10], [15], [38]. They are now used in many model-driven engineering environments (e.g., Ptolemy [2], BIP [7], AADL [19], and SysML [3]) to, for example, support interface and/or requirement verification (by model checking). Most of the methods initially developed to implement such processes faced quadratic complexity barriers resulting from a predominant reliance on specific models of automata [15], [35], and the lack of abstraction capability, reversibility and traceability. However, recent work such as real-time and scheduling contracts [10], [42] demonstrated the applicability of contract theory for reasoning on real-time constraints resulting from component analysis of software, hardware and control. Online scheduling and service contracts have also been considered in [4], [39]. These proved to scale to commercial automotive applications (e.g., patents US9459840B1 and US9477446B1) by relying on gradually coarser system-level abstractions of component behaviors, and by making verification problems amenable to ILP and SAT-solving. Additionally, architecture analysis based on contracts have been introduced to develop the use of such abstractions for causal reasoning across domains [25], [37].

Despite significant progress made by the aforementioned efforts, system design today is still confounded by a whole host of conflicting methods and tools that make systems non-compositional at multiple levels from physical construction to reasoning and validation. The challenges here include handling the large variety and number of abstractions, scalability, as well as the dynamic nature of computational elements. Time topology makes reasoning even more complicated as one must now account for differing views of time including event/causality in computing systems. Quasi-synchrony for embedded system design [8], [11], [16], delay- and latency-insensitivity for desynchronized circuits and systems [13], [33], precision-time computation (PRET [18]), Spanner's TrueTime [14], and Roseline's time service for cloud and grid infrastructures [17], have successfully established sound layered abstractions between noisy time measurements incurred by hardware variability and physical operating conditions, occasioning clocks drift and jitters, and system and software protocols to mitigate them and restore sound correspondences between logical software time, hardware real-time and physical hazards. In fact, both [10], [37] illustrate, in different ways, the inter-dependence of application, hardware, and physical constraints at the core of system integration, and demonstrate the impossibility of resolving them under strict separation and without integrative models of time.

Deep Specification. The pioneering SeL4 project demonstrated the barrier of a quadratic growth of proof obligations (in the size of code) [22], a barrier which every framework now seeks to avoid. The NSF DeepSpec project co-designed the specification and proof of the CertiKOS hypervisor in Coq [20]. It proved the applicability of deep specification and multi-layered abstraction in operating system design by building and proving an entire system stack in one heroic person-year. CertiKOS does not, however, state the cost to get composition of layered abstractions right, which is data of interest to system architects Such a methodology is fortunately common to abstract model checking: to define backtracking facilities to reverse or unwind a component's abstraction to a more concrete one [29], or contract-based design, should a component abstraction mismatch another it is to be composed with: causally locating the component to refine its interface [25].

End to End Verification. Boosted by full-scale verification projects such as miTLS [12], verified programming languages [36], [40] have demonstrated the ability to provide the modularity, multi-layered abstraction and proof capabilities compatible with a deep level of specification for system integration. RIOT, an operating system for IoT devices, also

integrates cryptographic modules verified with and generated from F* specifications [6]. Refinement type provide scalable logical abstractions and efficient proof automation tactics to gradually and compositionally model, compose, abstract, refine and prove systems built from variables, functions and modules typed by logical propositions with the potential to describe layered functions, modules, components, systems. Refinement type theory provides an unprecedented opportunities to bridge gaps across engineering domains by developing integrative verification and synthesis methods without breaking modularity.

Framework Design. Our main challenge is to develop a verified design methodology for CPSs that spans across the fields of hardware, middle-ware, software, signal processing and control design, across logical, discrete, continuous models of time. This goal is inspired from, and similar to, type-safe unikernels built using MirageOS [27] by the instantiatiation of parameterized system library modules written in OCaml.

Acknowledgments: This work is partially supported by Inria associate-project Composite and by Inria's International Chair program.

REFERENCES

[1] What is a motor resonance? http://www.motioncontrolguide.com/learn/faqs/motors/stepper-motors/what-is-stepper-motor-resonance, 2013.
[2] The ptolemy project. https://ptolemy.berkeley.edu, 2018.
[3] Systems modeling language. http://www.uml-sysml.org/sysml, 2018.
[4] Whip: safe composition of micro-services. ttp://whip.services, 2018.
[5] Arduino stepper motor. https://www.arduino.cc/en/Reference/Stepper.
[6] E. Baccelli, et al. Riot: an open source operating system for low-end embedded devices in the iot. *IEEE Internet of Things Journal*, pages 1–1, 2018.
[7] A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T. Nguyen, and J. Sifakis. Rigorous component-based system design using the BIP framework. *IEEE Software*, 28(3):41–48, 2011.
[8] G. Baudart. *A synchronous approach to quasi-periodic systems*. PhD thesis, PSL Research University, 2017.
[9] BBC. Police ground drones after reports they fall out of the sky. https://www.bbc.com/news/technology-46032019, October 2018.
[10] A. Benveniste, et al. Contracts for system design. *Foundations and Trends® in Electronic Design Automation*, 12(2-3):124–400, 2018.
[11] A. Benveniste, P. Caspi, P. Le Guernic, H. Marchand, J.-P. Talpin, and S. Tripakis. A protocol for loosely time-triggered architectures. In *International Workshop on Embedded Software*, pages 252–265. Springer, 2002.
[12] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and S. Zanella-Béguelin. Proving the tls handshake secure (as it is). In *International Cryptology Conference*, pages 235–255. Springer, 2014.
[13] L. P. Carloni, K. L. McMillan, A. Saldanha, and A. L. Sangiovanni-Vincentelli. A methodology for correct-by-construction latency insensitive design. In *Proceedings of the 1999 IEEE/ACM International Conference on Computer-Aided Design*, pages 309–315, 1999.
[14] J. C. Corbett, et al. Spanner: Gogle's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.
[15] L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the 8th European Software Engineering Conference*, pages 109–120, 2001.
[16] A. Desai, S. A. Seshia, S. Qadeer, D. Broman, and J. C. Eidson. Approximate synchrony: An abstraction for distributed almost-synchronous systems. In *International Conference on Computer Aided Verification*, pages 429–448. Springer, 2015.
[17] A. Dongare, P. Lazik, N. Rajagopal, and A. Rowe. Pulsar: A wireless propagation-aware clock synchronization platform. In *Real-Time and Embedded Technology and Applications Symposium*, pages 283–292. IEEE, 2017.
[18] S. A. Edwards and E. A. Lee. The case for the precision timed (PRET) machine. In *Proceedings of the 44th Design Automation Conference*, pages 264–265, 2007.
[19] P. H. Feiler, B. A. Lewis, S. Vestal, and E. Colbert. An overview of the SAE architecture analysis & design language (AADL) standard: A basis for model-based architecture-driven embedded systems engineering. In *IFIP TC-2 Workshop on Architecture Description Languages (WADL), World Computer Congress*, pages 3–15, 2004.
[20] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo. Certikos: An extensible architecture for building certified concurrent os kernels. In *OSDI*, volume 16, pages 653–669, 2016.
[21] Johnny five api - stepper. Online: http://johnny-five.io/api/stepper/.
[22] G. Klein, et al. sel4: formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 207–220, 2009.
[23] S. Kong, S. Gao, W. Chen, and E. Clarke. dreach: δ-reachability analysis for hybrid systems. In *International Conference on TOOLS and Algorithms for the Construction and Analysis of Systems*, pages 200–205. Springer, 2015.
[24] C. Lamb. Why is IoT so insecure? https://dzone.com/articles/why-is-iot-so-insecure, December 2016.
[25] T. T. H. Le and R. Passerone. Refinement-based synthesis of correct contract model decompositions. In *12th ACM/IEEE International Conference on Formal Methods and Models for Codesign*, 2014.
[26] S. Lunel, B. Boyer, and J.-P. Talpin. Compositional proofs in differential dynamic logic dl. In *International Conference on Application of Concurrency to System Design*, pages 19–28. IEEE, 2017.
[27] A. Madhavapeddy and D. J. Scott. Unikernels: Rise of the virtual library operating system. *Queue*, 11(11):30, 2013.
[28] S. Nakajima, J.-P. Talpin, M. Toyoshima, and H. Yu. Cyber-physical system design from an architecture analysis viewpoint.
[29] G. Plassan. Vérification formelle concluante des propriétés des systèmes multi-horloges. 2018.
[30] A. Platzer. The structure of differential invariants and differential cut elimination. *Logical Methods in Computer Science*, 8(4), 2011.
[31] A. Platzer and J. Quesel. Keymaera: A hybrid theorem prover for hybrid systems (system description). In *Automated Reasoning, 4th International Joint Conference*, pages 171–178, 2008.
[32] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57, 1977.
[33] D. Potop-Butucaru, Y. Sorel, R. de Simone, and J. Talpin. From concurrent multi-clock programs to deterministic asynchronous implementations. *Fundamenta Informaticae*, 108(1-2):91–118, 2011.
[34] J. Protzenko, et al. Verified low-level programming embedded in f. *Proceedings of the ACM on Programming Languages*, 1:17, 2017.
[35] J. Raclet, E. Badouel, A. Benveniste, B. Caillaud, A. Legay, and R. Passerone. A modal interface theory for component-based design. *Fundamenta Informaticae*, 108(1-2):119–149, 2011.
[36] P. M. Rondon, M. Kawaguci, and R. Jhala. Liquid types. In *ACM SIGPLAN Notices*, volume 43, pages 159–169. ACM, 2008.
[37] I. Ruchkin, D. De Niz, S. Chaki, and D. Garlan. Contract-based integration of cyber-physical analyses. In *International Conference on Embedded Software*, pages 1–10. IEEE, 2014.
[38] A. L. Sangiovanni-Vincentelli, W. Damm, and R. Passerone. Taming dr. frankenstein: Contract-based design for cyber-physical systems. *Eur. J. Control*, 18(3):217–238, 2012.
[39] I. Stierand, P. Reinkemeier, T. Gezgin, and P. Bhaduri. Real-time scheduling interfaces and contracts for the design of distributed embedded systems. In *8th IEEE International Symposium on Industrial Embedded Systems*, pages 130–139, 2013.
[40] N. Swamy, et al. Dependent types and multi-monadic effects in f. In *ACM SIGPLAN Notices*, volume 51, pages 256–270. ACM, 2016.
[41] J. van Rantwijk. Arduino clock frequency accuracy. http://jorisvr.nl/article/arduino-frequency, December 2012.
[42] H. Yu, P. Joshi, J. Talpin, S. K. Shukla, and S. Shiraishi. The challenge of interoperability: model-based integration for automotive control software. In *Proceedings of the 52nd Annual Design Automation Conference*, pages 58:1–58:6, 2015.
[43] H. Zhao, N. Zhan, D. Kapur, and K. G. Larsen. A "hybrid" approach for synthesizing optimal controllers of hybrid systems: A case study of the oil pump industrial example. In *FM 2012: Formal Methods - 18th International Symposium*, pages 471–485, 2012.