Special Session: Embedded Software for Robotics: Challenges and Future Directions

Houssam Abbas
University of Pennsylvania
Philadelphia, Pennsylvania
habbas@seas.upenn.edu

Indranil Saha
IIT Kanpur
Uttar Pradesh, India
isaha@cse.iitk.ac.in

Yasser Shoukry
University of Maryland
College Park, U.S.A.
yshoukry@ece.umd.edu

Rüdiger Ehlers
University of Bremen
Bremen, Germany
ruediger.ehlers@uni-bremen.de

Georgios Fainekos
Arizona State University
Tempe, AZ, U.S.A.
fainekos@asu.edu

Rajesh Gupta
University of California San Diego
California, U.S.A.
gupta@eng.ucsd.edu

Rupak Majumdar
MPI-SWS
Kaiserslautern, Germany
rupak@mpi-sws.org

Dogan Ulus
Boston University
Boston, MA, U.S.A.
doganulus@gmail.com

ABSTRACT
This paper surveys recent challenges and solutions in the design, implementation, and verification of embedded software for robotics. Emphasis is placed on mobile robots, like self-driving cars. In design, it addresses programming support for robotic systems, secure state estimation, and ROS-based monitor generation. In the implementation phase, it describes the synthesis of control software using finite precision arithmetic, real-time platforms and architectures for safety-critical robotics, efficient implementation of neural network-based controllers, and standards for computer vision applications. The issues in verification include verification of neural network-based robotic controllers, and falsification of closed-loop control systems. The paper also describes notable open-source robotic platforms. Along the way, we highlight important research problems for developing the next generation of high-performance, low-resource-usage, correct embedded software.

CCS CONCEPTS
• Computer systems organization → Embedded systems; Robotics;

KEYWORDS
Embedded software, Robotics, Neural networks, monitor synthesis, Robot Operating System, Secure state estimation

ACM Reference Format:


1 INTRODUCTION
There is a trend towards increased and higher-level autonomy in robotics. The trend is most evident in mobile robots, such as self-driving cars and Unmanned Aerial Vehicles (UAVs), but it also affects personal robotics, warehouse robots (e.g., Kuka robots), and other application domains such as medical devices. These robots are tasked with understanding the world around them, planning their actions in it, and more often than not, interacting with the humans that also occupy that world.

The increased autonomy has two immediate correlates, which form the basis for the issues we address in this paper. First, the robots’ missions are increasingly complex, whether on the perception or action side, and go well beyond traditional objectives that were restricted, for example, to pre-defined movements, or static environments, or very low-level scene perception sufficient for these limited tasks. Secondly, because these robots are operating at human scale (on the roads, in homes and in warehouses) and performing dangerous missions (like driving, mine inspection or surgery), the correctness requirements are stringent and must be proved to hold. Contrast this with the more traditional approach in which satisfactory performance was established through testing and experimentation, and boiled down to a best effort (albeit a gargantuan one in some cases). The algorithms that govern these robots, therefore, are pushing the limits of the theories and tools at our disposal for checking functional correctness in a timely fashion, which complicates the design and verification phases. The resulting code is also prompting the use of ever-higher performance hardware and software architectures, and software components (like neural nets). However, these are also much harder to analyze rigorously, thus complicating the implementation phase. Finally, it remains the case that software is best tested on the target hardware and in the target software stack to reduce deployment surprises, and this prompts the creation of realistic yet accessible robotic platforms.
This paper surveys the above issues and describes recent approaches to addressing them. In Section 2, we describe the design challenges, focusing on the recent correct-by-construction paradigm. We illustrate the potential of control synthesis by describing an approach to the automatic generation of correct motion planners. We also describe the design of correct-by-construction runtime monitors targeting ROS-based software, and the problem of designing secure state estimators.

In Section 3, we describe implementation challenges. We first tackle the problem of generating controller code that still meets the guarantees offered by the continuous-space algorithm. Then, we focus on the issues that arise in using Commercial Off-The Shelf (COTS) heterogeneous architectures, including ill-specified Graphics Processing Units (GPUs), for safety- and timing-critical applications, and in using the under-specified OpenVX standard for real-time computer vision applications. We conclude the section with a discussion about the creation of efficient implementations of neural network code.

Section 4 addresses verification challenges. It describes approaches to verify the correctness of network-based controllers, and a general testing framework for embedded software that includes neural network components. Finally, Section 5 surveys a few notable open-source platforms that provide the complete instructions for building, programming and using reduced-scale self-driving cars, humanoids and manipulator arms.

2 DESIGN OF ROBOTIC SOFTWARE

A major impediment to developing large-scale robotic infrastructure with high-level autonomy is the lack of systematic design and programming support. Autonomous robotic systems cut across many layers of the system stack: from low-level physical dynamics, sensing, control, and scheduling, to high-level software systems issues such as goal specification, co-ordination, provisioning, and fault tolerance. While different languages, IDEs, and APIs provide support at different layers (e.g., Simulink and Stateflow for dynamics modeling, ROS [76] for messaging, etc.), the programmer is often left to navigate the zoo of languages, systems, and methodologies. In this section, we will discuss how high-level specification and formal methods-based automated synthesis can help us develop a unified framework for solving complex problems in the robotics domain.

2.1 Systematic Design and Programming Support

Traditionally, the specification for a mobile robot has been the point-to-point reachability with obstacle avoidance. In the last decade, Linear Temporal Logic (LTL) [8] has been widely used to capture complex specifications for the mobile robots and various methodologies have been used to synthesize trajectories automatically from the specification. In one of those methodologies, a finite model for the robot dynamics is first generated using an abstraction process based on discretization of the configuration space [16], and then game-theoretic or model checking-based synthesis techniques are used to generate high level motion plans and low level control policies on the abstract model [17]. In another approach, SMT Solvers or Mixed-Integer Linear Programming tools are used for the composition of motion primitives for the robots to synthesize trajectories satisfying a given specification [47, 66, 80, 81, 98, 101].

In recent years, several systems have been designed to provide a more uniform set of abstractions for multi-robot systems [28, 41]. These systems typically provide a high-level programming model to specify tasks and a compiler and runtime system that compiles the high-level tasks into individual plans executed by each robot. The aim is to close the semantic gap between the declarative specification of tasks at the level of the user and the low-level details of managing individual distributed mobile robots, scheduling and planning, etc. The analogy is with the cloud [24]: we expect the cloud infrastructure to abstract away low-level details of the software stack, allowing a high-level view of the application.

For example, Antlab [41] provides a declarative task specification language based on linear-time temporal logic (LTL). The programming model of Antlab represents the underlying world as a discrete abstraction of physical space together with a set of predicates and provides an abstraction for the set of available robots. The user does not program individual robots or even know how many robots there are; instead, the user knows a set of action primitives the robots can perform, and declaratively specifies a desired temporal sequence of actions. The propositions in a task can range over spatial locations (“reach location ℓ”) as well as action primitives (“pick up”, “drop”) and the temporal connectives allow expressing application-level behaviors over time. The quantification over robots allows the programmer to specify a task without referring to individual robots but also helps express co-ordinated behaviors (“two robots follow each other”). Specifically, the user does not need to know about current states of the underlying robots; it is the responsibility of the run-time system to figure out which robots to assign to a task, how to schedule and plan the task, how to co-ordinate robots, and how to ensure the system has high throughput.

The compiler for these systems implement a combined task and path planner which gets as input a batch of user tasks and produces optimal paths for a group of robots such that all the user tasks are completed, if possible [57, 93]. The planning algorithm can be implemented using an SMT solver (such as Z3 [27]) or an AI planner supporting LTL constraints [70]. The plan for each robot is implemented on top of the robot’s dynamic navigation stack. This allows taking into account dynamic uncertainties in the robotic environment, for example, dynamic obstacles or imprecision in actuation, and implementing the plans in a receding-horizon style, where deviations from an ideal plan are monitored and, if necessary, re-planned. Finally, a software services layer provides services such as provisioning and fault tolerance.

While these systems are an important first step, their application is still limited to simple “warehouse style” tasks with a fixed and usually small vocabulary of tasks. Currently, the planners in these systems “string together” a fixed set of motion primitives. In future systems, we expect to see a richer language of task specifications as well as more scalable task and path planners, and one can imagine more expressive capabilities programmed on the robots in almost real-time. Finally, co-ordination between different robots are rather limited in current systems. On the other hand, recent advances in reinforcement learning techniques show that autonomous agents can learn very expressive behaviors [49]. It will be interesting to
see systems which can incorporate such expressive behaviors and yet retain the simplicity of the programming abstraction and allow end-to-end reasoning.

2.2 ROS-based design and monitor synthesis

Good design practice requires the creation of runtime monitors, which are pieces of code that can monitor key properties of the system’s behavior in real-time, report any violations, and possibly enforce fail-safe behavior. Simple monitors to watch resources and detect local faults are prevalent in robotic applications. However, with the increased requirements in perception and control, current robots and autonomous systems necessitate more complex monitoring tasks during their operation. These complex monitoring tasks range from enforcing safety and security properties and ensuring the correct execution of synthesized plans, to pattern matching over sensor readings to help perception. A promising direction is to generate these complex monitors automatically from their high-level specification and integrate them into ROS-based design1. In the following, we survey high-level monitor specification languages, the generation of such complex monitors from these specifications, and ROS-based tools.

Regular expressions and temporal logic are two major specification languages to describe temporal patterns and properties. These formalisms provide powerful and well-studied frameworks to specify temporal order and concurrency among various events and states. Their many variants and extensions have been proposed to address different aspects of complex monitoring tasks [12, 45]. In particular, timed, quantitative, and parametric extensions of regular expressions and temporal logic are very interesting for robotic applications. For example, a pattern such that the value of a sensor X is below 4.2 for 10 seconds and then the value of the same sensor X is above 4.2 for 5 seconds involves timing constraints, numerical comparisons, and variables that can be handled by these extensions.

Monitor generation from the high-level specification can be divided into three approaches. In the first approach, the monitor is designed to rewrite the original specification according to a set of rules and the current input [78, 96]. Then, the monitor alerts if a certain form has been obtained after the transformation. In the second approach, the monitor is designed as an automaton, which is essentially a big look-up table that contains all possible transformations for all possible inputs. Since the table is pre-computed, the performance at runtime is considerably better than rewriting-based monitors. However, this automata approach does not scale well for the quantitative and timed extensions we mentioned above and automata-based monitors suffer from severe limitations [11, 64, 65]. Specifically, the automata are potentially very large, are non-compositional and non-extensible.

In the third approach, the monitor is designed to be a network of small computation nodes. By its nature, this approach is compositional, extensible, and offers several other theoretical and practical advantages [72, 95]. Although this idea emerged very early in [23] for regular expressions, the network approach was not exploited much until the paper [46] in which authors essentially propose

network-based monitors generated from temporal logic specifications. Subsequent works have extended this approach for timed specifications [15], quantitative [29], and parametric [14, 44].

These approaches and algorithms have been implemented in several standalone tools such as [6, 13, 65, 77, 94]. However, such fragmentation of tools, interfaces, and programming languages makes monitoring a challenging technology to use in robotics. An important achievement would be in developing an extensive framework that handles specifications in a unified manner and generates monitoring ROS nodes to be deployed in robotic applications. The first project in this direction was ROSMOP2, which supports a subset of MOP software monitoring framework [65]. More recently, the tool REELAY3 has been proposed to generate network-based monitors with several practical enhancements and ROS support. As these tools make complex monitoring tasks more accessible in robotics, we would see them integrated in perception and control algorithms more often in the future.

2.3 Design for security: secure state estimation

The active nature of robotics, where data collected from various sources are then used to make decisions and actions, opens the door to new attack vectors, based in the physical world, that can be extremely damaging. Classical cybersecurity countermeasures are oblivious to such attacks. For example, if the adversary manipulates physical/analog signals before digitization [84, 87], no amount of digital security can help. It’s unsurprising, then, that a multitude of fatal and life-threatening situations can be created by such attacks as demonstrated by the recent sensor spoofing attacks on various automotive and robotic platforms [84, 87, 102].

A very recent security trend is the exploitation of the continuous dynamics of the robot to provide security [22, 39, 69, 86]. That is, by using an accurate mathematical model for the underlying physics of the robot, one can explain any discrepancy between the measured sensor data and the expected measurements—as per the model—as being the result of an adversarial attack. Once the malicious sensors are detected and isolated, one can estimate the state of the underlying physical system by using the data collected from attack-free sensors. This technique is referred to as secure state estimation.

Detecting and mitigating attacks on sensory data is, in general, a combinatorial problem [69], which has been addressed either by brute force search, suffering from scalability issues [22, 69], or via convex relaxations using algorithms that can terminate in polynomial time [39, 86] but are not necessarily sound. However, the computational performance as well as the security guarantees can be improved by leveraging results from formal methods literature which lead to building an satisfiability-module-theory (SMT) engines that is particularly tailored towards the secure state estimation problem [85].

3 IMPLEMENTATION OF ROBOTIC SOFTWARE

Preserving the theoretical correctness guarantees of an algorithm post-implementation requires paying careful attention to the effects

---

1 Robot Operating System (ROS), the de facto standard middleware for developing robotic software. See www.ros.org.
2 https://github.com/Formal-Systems-Laboratory/rosmop
3 https://github.com/doganulus/reelay
of finite precision arithmetic and other implementation imprecisions. The choice of target platform that runs the code fast enough and within the power envelope is a crucial decision. Some algorithms that are expensive in energy or memory, like neural net inference, might need to be approximated, at the software or hardware levels, to make them run on the target platform. Standards can help developers get a handle on an implementation’s characteristics, like timing bounds. This section looks at each of these issues in turn.

### 3.1 Synthesis of Controller Software

The feedback controller for a robotic system is designed using real arithmetic, considering the dynamics of the system to be continuous. A mathematical analysis ensures the correctness of the designed controller. However, when the controller is realized as software, the dynamics of the system are discretized based on some chosen sampling time, and finite precision arithmetic is used to represent the variables. Now, one should ensure that the software implementation of the controller still satisfies the desired properties guaranteed by the continuous-space design. One main property of interest for robotic systems is practical stability or region stability [73], which requires the controller to steer the output of the dynamical system to a region around the desired value. Anta et al. [7] show that the verification of the region stability property for the implemented control system can be reduced, through a control-theoretic analysis, to the problem of computing a bound on the error due to quantization effects introduced in the implementation of the controller program. Now a program analysis technique can be employed to calculate a bound on the implementation error. For the computation of the error, Anta et al. employ a static program analysis technique which is based on verification condition generation [100]. It reduces the error bound computation question to a validity problem for a formula in the combination theory of reals and bit-vectors, for which off-the-shelf, efficient decision procedures are available [33].

One can leverage the error bound computation technique for controller implementation to address the following control software synthesis challenge: for a chosen finite precision arithmetic, design a controller for which the implemented software has the least error among all possible controllers. Generally, controllers are designed to minimize the control cost (the power of the control signal) and the state cost (the deviation of the state from the desired value). It can be shown that the controller designed based on optimization of such costs may produce a controller whose finite-precision implementation has a significant error in the output. Majumdar et al. [63] employ a stochastic optimization-based methodology [55] to synthesize a feedback controller that minimizes both the state and control cost along with the error in the finite-precision implementation of the controller software.

The feedback controllers for a dynamical system usually have the form of a linear expression (for linear control systems [53]) or a polynomial (for nonlinear control systems [56]). A naive compilation of a controller expression may produce a program whose output may significantly deviate from the controller designed using real arithmetic, whereas a different order of evaluation may result in a program producing outputs that are close to the values of the real-valued expression on all inputs in its domain. One can devise a compilation scheme to compile the arithmetic expressions for the controllers to fixed-point arithmetic programs by finding an optimal ordering of the arithmetic operations. Darulova et al. [26] present such a compilation scheme that minimizes the error in the controller program using fixed-point arithmetic with respect to a naive compilation of the real-valued expression. The presented technique is based on genetic programming [74], where the fitness of each candidate program is determined based on the bound on the error at the output of the program. To compute the bound on the error, they employ a static analysis technique based on affine arithmetic. Recently Darulova et al. [25] have demonstrated that assigning different precision to different variables may lead to further improvement in the accuracy of the program.

Despite the progress made on the verification and synthesis of controller software for linear systems and simple nonlinear systems, generalizing the approaches to complex robotic systems is still challenging due to two main reasons. First, the verification of the closed-loop control system with respect to a stability specification relies on a control-theoretic analysis that reduces the verification problem to a program analysis problem. The control-theoretic analysis is based on Lyapunov function(s) [56] that establish the stability of the closed loop. Most controllers for practical robotic systems are nonlinear in nature. Synthesizing a Lyapunov function for such systems is often very challenging. Recently introduced neural network-based controllers for dynamical systems (e.g. [48]) hardly come with any stability guarantee in the form of a Lyapunov function, resulting in the infeasibility of verification of the software implementation of such controllers. Second, the error analysis of a controller program relies on abstract interpretation [42] or reduction to an SMT problem [10]. For programs involving nonlinear computations, both these approaches are unsatisfactory in terms of either precision (in case of abstract interpretation) or scalability (in case of SMT solving).

### 3.2 Real-time platforms for safety-critical robotics

The increased levels of autonomy planned for vehicles and personal robots require computation-intensive perception and planning algorithms. Multi-core, heterogeneous computer architectures that use Commercial Off-The Shelf (COTS) components can meet the performance requirements of new applications. However, it is difficult to provide predictable timing guarantees when several cores contend for shared resources, like memory or buses. Safety-critical robotics applications (like navigation in a self-driving car) require such timing guarantees in order to provide reliable performance and assured safety. Thus an important research question is the development of scheduling algorithms and corresponding analyses for COTS architectures, perhaps customized to common or critical algorithms like path planning.

In this context, the European Hercules project [19] studies the suitability of existing hardware architectures, programming models and real-time operating systems for safety-critical applications on heterogeneous architectures, and should provide a wealth of hard data to guide future research. E.g., in [40], a state-of-the-art path planner is implemented on the Jetson TX1 with the Predictable Execution model (PREM), which separates programs into memory and compute phases that can be independently scheduled. It is
shown that this reduces the Worst-Case Execution Time (WCET) of the application, and reduces the sensitivity of DRAM accesses to CPU interference. Similar studies for other path planners and online monitoring code, to name a few, will enable a more reliable deployment of embedded code. In particular, it will be interesting to study the trade-offs for monitoring code between correctness of output and timing-predictable execution.

A second issue, which arises when using Graphics Processing Unit (GPUs) boards for computation-intensive tasks (e.g., data-parallel inference or computer vision algorithms), is their proprietary nature and scant documentation about their scheduling [104]. The survey paper [104] provides a valuable compendium of pitfalls when trying to ensure temporal correctness of GPU applications, with an emphasis on autonomous driving. Alleviating these issues should serve as a source of research problems. The ‘opaque’ nature of GPU scheduling also makes it necessary to first infer their behavioral rules and use these to estimate their worst-case timing behavior. E.g., scheduling rules were inferred in [5] for NVIDIA’s GPUs. However, any such inferred rules might be invalidated by future changes to the hardware [9]. A long-term goal would be to automate rule generation and validation in a more quantitative manner, such that rule violations can be ranked and their impact on a given application used as a guide for hardware choice and programming.

Finally, we mention the ever-present issue of power consumption: these new algorithms for autonomy are power-hungry. A rough estimate based on publicly available data [68] indicates that a small electric vehicle can have its drive time reduced by almost 62% if it drives in autonomous mode. Profiling and reducing this power consumption is essential, especially in domains, like the automotive industry, where emissions are regulated and every Watt counts. A reduction of power consumption will also decrease the cost of the systems as they will use smaller batteries and less expensive computing boards. Here, a joint optimization of perception and control tasks can lead to power reductions [20, 68] and enable smarter resource usage.

3.3 Efficient implementation of neural network inference

A software artifact of special importance is Neural Networks (NNs): their impressive successes in perception applications, like object detection, makes them a natural choice for mobile robotics. However, their energy consumption and memory usage limit their use to high-end GPU+CPU platforms, which might not be an option for lighter or small form factor robots. E.g., convolutional NN AlexNet uses 61 million parameters, 239 MB and 1.5 billion FLOPs. Across the hardware types that neural nets have been implemented on, from FPGAs to SoCs (NVIDIA Tegra or Samsung Exynos) to custom super-computers (like DaDianNao [21]), energy and memory are important implementation criteria [89]. The creation of efficient (low-power, low-memory) implementations of forward propagation, which is the operation that typically takes place on-board the robot, is thus an important research problem in embedded robotic software. As a starting point for researchers entering this field, a tool for estimating a NN’s power consumption is available online⁴. Methods to reduce energy and memory usage in NNs can be categorized into methods that do not impact accuracy (i.e., the original and optimized NN produce bitwise identical results) and those that sacrifice some accuracy for further energy or memory savings. The survey [89] gives an excellent overview and comparison of these two sets of methods, and in particular of data flow optimizations. Here, we complement that survey with more recent results that fall in the second, ‘lossy’, category.

Approximate computation is a general paradigm in which computations are performed approximately at a lower energy cost [50], and has naturally been applied in NNs to perform approximate Multiply-Adds. The work in [51] introduces a new way to do approximate computation through the identification of opportunities for computation reuse, then exploiting these opportunities in an approximate, energy-efficient manner. Specifically, frequent input patterns to a network’s layers are experimentally identified, and their computation results stored, so they can later be retrieved (via approximate matching), thus saving the cost of re-performing the operation time and again. Results range from a 22.3% energy savings (with a NN accuracy of 98.3%, from a baseline accuracy of 98.5%) to 58.9% energy savings (with a NN accuracy of only 60.6%). Moreover, the matching approximation degree is configurable online, and it would be interesting to devise a ‘scheduling’ algorithm for switching between approximation modes.

Binarization of a net binarizes the weights and activations of the NN, thus turning the (expensive) multiplication into a boolean operation. The recent work on Local Binary Pattern Networks [61] (LBPNets) extends this idea by learning the binary pattern as part of end-to-end supervised learning, as opposed to classical binarization which uses a fixed, non-optimized pattern. This is followed with dimension reduction by random projection. By replacing convolutions by logical operations (comparisons), LBPNets save energy: at the 45nm node, a comparator uses less than 3e-14 J, while a 32 bit multiply-adder uses at least 3.7e-12 J [61]. More generally, overall size and latency are significantly reduced relative to a full Convolutional NN, at the cost of a modest reduction in accuracy. While LBPNets are targeted to so-called edge devices (e.g., sensors with some computational power), it would be interesting to explore their applicability to more demanding applications. A different extension of binarized networks uses binarized separable filters to perform the neuronal operations [60], thus reducing the model size. Corresponding training algorithms are developed, at the cost of a mild accuracy decrease compared to the original binarized CNN.

3.4 The OpenVX standard for computer vision applications

Vision-based sensors are widely used for robot navigation as cameras are cost-effective sensors to perceive the environment. Recently, the Kronos Group has introduced a ratified standard named OpenVX [43] to facilitate the development of real-time embedded applications based on computer vision techniques. In the OpenVX environment, computer vision computations are represented as directed graphs. The nodes in the graph represent the vision-related functions, and the edges capture the precedence and data dependency among the tasks. Though OpenVX can be applied on various

---

⁴At http://eyeriss.mit.edu/energy.html.
platforms, GPUs have been the most popular platform to implement OpenVX applications.

On safety-critical robots, the implementation of the vision algorithm needs to satisfy strict real-time constraints to ensure end-to-end latency in the control loops. OpenVX does not currently provide enough support for hard real-time computations. For example, concepts like priorities and graph invocation rates that are essential for real-time applications are missing from the standard. Moreover, the computation associated with a graph is expected to get executed monolithically, which hinders the exploitation of such parallelism.

Researchers have attempted to create a modified version for OpenVX which will explicitly address the real-time requirements of the vision-based applications. E.g., Elliot et al. [36] and Yang et al. [52] treated the nodes in the graph as schedulable entities which allowed more parallelism, leading to an improvement in the bound on the response time. In a recent work, Yang et al. [103] have developed a fine-grained representation of OpenVX graphs, which provides further scope for parallelizing the computations in a vision application. In this model, the computation corresponding to a node in the graph is further subdivided into a finer set of computations, and also the consecutive jobs corresponding to the same task are allowed to execute in parallel. In a case study on a computer vision application for pedestrian detection using six cameras, the authors in [103] demonstrate that the fine-grained model can guarantee a bounded response for all the six cameras, whereas the coarse-grained model can support only one camera.

The remedying of these and other deficiencies can stimulate future work in real-time standards for vision applications, and more fine-grained models of the tasks for computer vision applications can be expected to be part of the OpenVX standard in the future, enabling its widespread adaptation in the robotics community.

4 VERIFICATION AND TESTING OF ROBOTIC SOFTWARE

The reported failures of complex robotics software in safety-critical situations, sometimes leading to human fatalities [1], emphasize the continuing need for more powerful methods to test and verify the software, and underlying algorithms. We distinguish between two broad categories for checking the safety (and more generally, the correctness) of a system: the first is Testing, in which the system (or a model of it) is simulated N times following some simulation strategy, and the outcome of these simulations is taken to be indicative (in a more or less formal sense) of the true correctness of the system. Note that testing is usually incomplete: failure to find a bug in N simulations does not rule out that a bug does exist.

The other category is formal verification methods, which performs automated reasoning on a mathematical model of the system to yield exact results. That is, unlike testing, if the system can produce an incorrect behavior, then a formal verification method will find it (completeness), and if the method does return an incorrect behavior, then that is indeed an error in the model (soundness)\(^5\). Section 2.1 described the application of formal methods to the control synthesis problem, and Section 3.1 to the verification of a controller code. This section focuses on an emerging area in the checking of robotic software, specifically, the verification and testing of neural network models and of systems containing neural networks. This clearly has applications beyond embedded software.

4.1 Verification of neural network-based robotic controllers

Advances in designing robotic controllers based on machine learning components has created an urgency to study their safety and reliability [58, 59, 82, 83]. Several works have been reported in the last few years attempting to apply formal verification techniques to machine learning components in general, and neural networks, in particular. The work in this area can be classified into two categories: (i) component-level and (ii) system-level verification.

In general, verifying formal properties of feed-forward neural networks is a challenging task because the number of their parameters is very large (several millions). Recent works focused on specific NN architectures that are amenable to verification using Satisfiability Modulo Theories (SMT) solvers and Integer Linear Programming (ILP) solvers. In the first class of recent works—component-level verification—researchers focused on verifying NNs against input-output specifications when the NN nonlinearity is restricted to be a piecewise affine function known as the Rectifier Linear Unit (ReLU) [18, 34, 35, 54, 79]. Such input-output techniques compute a guaranteed range for the output of a deep neural network given a set of inputs represented as a convex polyhedron. A central difficulty in such a task is to consider all possible phases of all ReLU nonlinearities, which is daunting given that NNs can have thousands of ReLUs (A network with n ReLUs has \(2^n\) phase combinations). A common technique is to relax the ReLU nonlinear function to a linear/convex function. This relaxation allows to quickly rule out large combinations of the ReLU phases that will not violate the input-output specifications [18, 35].

To circumvent the drawback of using simple input-output range specifications and reason directly about system safety, the second class of recent works focused on finding corner-cases that lead to the violation of system-level safety specifications. To that end, recent works focused on testing and semi-formal verification (e.g., falsification) [31, 62, 71, 88, 90, 97, 99, 105]. In these works, the objective is to generate several scenarios which trigger different parts of the NN. Different algorithms are proposed for generating these scenarios including random sampling [31], generative adversarial networks [105], and node coverage [71, 90]. These approaches might scale to larger models, but we are still a long way from tackling the most successful network architectures in use today.

4.2 Falsification of the closed-loop control system

When the system model of interest falls in a class that cannot be handled by formal verification, or the model size grows too large, one has to resort to testing. This is currently necessary for embedded control systems that incorporate a perception NN: the size of the latter makes it well outside the scope of formal methods. This section describes recent approaches to the testing of self-driving

\(^5\)Approximate formal methods might not be sound (e.g. over-approximate reachability may find spurious errors). They are necessary, for example, to deal with systems with general continuous dynamics, but we don’t emphasize this distinction here.
cars, as a prototypical example of control-perception NN system. They can be divided as follows:

- approaches that test the control sub-system separately from the perception, under a bounded error model on the perception’s output [38, 67, 92]. In essence, this is the ‘traditional’ setup for testing of embedded and cyber-physical systems.
- approaches that test the control-perception jointly, and the testing is guided by the control objectives, which is ultimately what matters (rather than just finding perception errors that don’t affect the control) [4, 91].
- approaches that test the two sub-systems separately, but couple the two tests as described below [32].

Almost all of the above leverage a testing technique known as Robustness-Guided Testing (RGT) [2, 37]. Briefly, in RGT, the system’s correctness is specified as a formula in a temporal logic, and the satisfaction of the formula by a finite-duration system execution $x$ is encoded in a real-valued function $f$ such that $f(x) < 0$ implies that $x$ violates the formula. Thus, the search for violations, or falsifiers, can be done by minimizing $f$ over the space of finite-duration executions $x$ [2, 6, 30]. This approach is broadly applicable since it only requires the ability to obtain system executions, so even black-box systems can be tested.

The works in [38, 67, 92] do not run a perception pipeline in the test loop. In [67], the output of the perception sub-system is modeled as a noisy state estimate with known error bound. The control sub-system is modeled as a hybrid dynamical system and a combination of SMT-solving (a formal method) and RGT, developed in [3], is applied to quickly find control errors. This approach is applicable to perception tasks that have a continuous output for which we can define a useful measure of error, like localization error; it will be useful to extend it to discrete outputs (like binary object detection).

Developing new ways of deploying this combination of exhaustive verification and RGT will be useful in achieving further speedups. In [92], a similar stochastic optimization setup is used to search for collisions between vehicles. In [38], reachability (another formal method) is combined with local sensitivity analysis of black-box models to find errors in Automatic Emergency Breaking, and assess the risk of the automotive components. This work was extended with control synthesis and implemented in the tool DryVR\(^5\).

In [4], RGT is applied directly to the problem of finding errors in a photo-realistic simulation of a self-driving car. The car detects objects in its video feed using the YOLO NN, and uses a lattice planner with a low-level PID tracker to navigate a T-junction. Results show that in this context, RGT can find a combination of starting position and velocity of the cars at the T-junction, and a time of day, such that the object detector NN will make certain errors that lead to a collision. The work in [91] also applies RGT to the falsification of a self-driving car system’s simulation, and further develops accelerations based on covering arrays, and compares different optimization strategies for this problem. In this area, finding good optimization heuristics that may be tuned to the NN structure is a useful problem to pursue.

The approach in [32] divides the falsification task in two: first, run RGT on the control sub-system with the NN output fixed to pessimistic (always wrong) and optimistic (always right) values, and find bugs in both cases; and secondly, search (an abstraction of) the input space of the NN\(^7\) to see whether it can make a mistake that activates the control bug. Current results indicate the usefulness of this approach for finding bugs. It would be useful to extend this decomposed approach to handle multi-frame perception errors and to relax the optimistic/pessimistic assumptions.

In all the above approaches, the question remains about the value of testing the NN with synthetic data (here, synthetic video). While [4] studies this question from both an algorithm-specific and algorithm-agnostic way, there is much computer vision work to be done to answer this question.

## 5 OPEN-SOURCE PLATFORMS

Ultimately, embedded software needs to be tested and profiled on the target hardware that it is meant to run on. The approaches to testing and verification surveyed above go a long way towards reducing and bounding surprises at deployment time, but these cannot be eliminated, as they are caused by a wide variety of factors that invalidate the assumptions made during the test efforts. Such factors include the rest of the software stack, the quality of the sensors, actuators, and communication infrastructure, and the physical environment. This can be an impediment to academic research groups that may be experts in one area of embedded software development, but not the entire process of building a functioning, useful robot. Open-source robotics platforms provide a solution to this very problem. An open platform is like a complete recipe for building, programming and using a robot, like a self-driving car, from scratch. In this section, we survey some open platforms that are well-supported, relatively affordable, and extensible. There are certainly others, but here we focus on platforms that are simple enough that most engineering students can build them, yet feature-rich enough that they present most of the common challenges faced in real-world deployment.

The first platform is the F1/10 autonomous race car (f1tenth.org), which is derived from the MIT Racecar\(^8\). F1/10 is actually three efforts: a) an open-source 1/10th-scale autonomous race car, b) a set of educational materials on basic concepts of perception, planning and control, currently being built into a full semester course, c) and a yearly racing competition open to teams from around the world. Notable features of F1/10 are its use of a powerful mechanical chassis that provides realistic physics for testing navigation and control algorithms; the ROS-based software stack that is easily extended (e.g., teams wishing to use a camera instead of the $1,500 LiDAR can do so); and the F1/10 simulator that allows algorithm testing within the deployment software stack and on the target hardware. The platform can also be used for formal verification and online monitoring research, and current work seeks to create a Runtime Monitoring ROS package which uses the REELAY tool mentioned in Section 2.

The Berkeley Autonomous Race Car (barc-project.org) is an open platform for autonomous driving whose focus is on control design and cloud data collection. Built on the same mechanical chassis as F1/10, it uses the Odroid XU4 as computer, which is less powerful than the Jetson TX1/TX2 used in F1/10.

---

\(^5\)See https://github.com/qibolun/DryVR_0.2.

\(^7\)The input space is the set of single frames around a given frame.

\(^8\) https://mit-racecar.github.io/
At a lower price point and with more computational and mechanical limitations, we mention two platforms: the first is TurtleBot3 from Robotis and Open Robotics. It is a small dual-wheeled mobile robot. It features a Raspberry Pi 3, a Robotis OpenCR board, and touch, infrared, color, and IMU sensors. The second is Duckietown (ducktownt.mit.edu), which is the least expensive mobile platform at $150, and has a nice ‘fleet mode’ to easily enable multi-robot scenarios.

Finally, we mention Poppy (www.poppy-project.org), a platform for interactive robotics, which offers instructions for building (or purchasing parts of) a humanoid or manipulator arm. Poppy has an emphasis on modularity of construction and versatility of applications.

6 CONCLUSIONS
This is an exciting time for researchers in the domain of embedded software for robotics. The increased demands placed by autonomy on the hardware and software of robotic platforms leads to rich new areas of research in all development phases, from design to deployment and beyond. This has led to a natural, yet challenging, convergence of ideas from artificial intelligence, control theory, formal methods, digital and analog design and software engineering. This survey highlights the most salient challenges that arise and recent approaches to tackling them. Much remains to be done.

REFERENCES


