Programming Safe Robotics Systems: Challenges and Advances

Ankush Desai[1(✉)], Shaz Qadeer[2], and Sanjit A. Seshia[1]

[1] University of California, Berkeley, USA
ankush@eecs.berkeley.edu
[2] Microsoft, Redmond, USA

Abstract. A significant challenge for large-scale deployment of autonomous mobile robots is to program them with formal guarantees and high assurance of correct operation. Our approach towards enabling safe programming of robotics system consists of two parts: (1) a programming language for implementing, specifying, and compositionally (assume-guarantee) testing the high-level reactive robotics software; (2) a runtime assurance system to ensure that the assumptions used during design-time testing of high-level software hold at runtime. Combining high-level programming language and its systematic testing with runtime enforcement helps us bridge the gap between software testing that makes assumptions about the low-level controllers and the physical world, and the actual execution of the software on a real robotic platform in the physical world. We implement our approach in Drona, a programming framework for building safe robotics systems. This paper introduces the Drona toolchain and describes how it addresses the unique challenges involved in programming safety-critical robots.

1 Introduction

Autonomous robotics systems have diverse and safety-critical roles in society today, including delivery systems, surveillance, and personal transportation. This drive towards autonomy is leading to increasing levels of software complexity. To tame this complexity and ensure safe and reliable operation of robotics systems, we have developed tools and techniques for programming and reasoning about them. In this paper, we present an overview of our work.

At the heart of an autonomous robot is the specialized onboard software that must ensure safe operation without any human intervention. The robotics software stack usually consists of several interacting modules grouped into two categories: *high-level* controllers, taking discrete decisions and planning to ensure that the robot safely achieves complex tasks, and *low-level* controllers, usually consisting of closed-loop controllers and actuators that determine the robot's continuous dynamics.

The high-level controllers must react to events (inputs) from the physical world as well as other components in the software stack (Fig. 2). These controllers are therefore implemented as concurrent event-driven software. However,

T. Margaria and B. Steffen (Eds.): ISoLA 2018, LNCS 11245, pp. 103–119, 2018.
https://doi.org/10.1007/978-3-030-03421-4_8

such software can be notoriously tricky to test and debug due to nondeterministic interactions with the environment and interleaving of the event handlers. We advocate writing the high-level controller in our domain-specific language P [1] which is suitable for expressing not only the asynchronous event-driven computation in the controllers but also models of other software components and the physical world (used for analysis/testing). By expressing both the controller and its environment in a single programming language, we bring advanced specification and testing techniques incorporated in P to the domain of robotics software.

Real-world robotics systems are rarely built as a monolithic system. Instead, they are a composition of multiple interacting components that together ensure the desired system specification (e.g., our case study in Fig. 2). P supports a module-system for the compositional reasoning of such complex event-driven software [2,3]. The P module-system enables implementing each component of the robotics software stack as a separate module and perform scalable compositional testing based on principles of assume-guarantee reasoning [4–6]. Compositional testing provides orders of magnitude more test-coverage compared to the traditional monolithic systematic testing approaches [2], uncovering several critical bugs in our implementation that could have caused safety violations.

Since the physical world often exhibits non-linear behavior, discrete-state models of the physical world used when testing the high-level controllers are necessarily approximate. Similarly, discrete-state models of the low-level controllers, which often use machine learning techniques, are also approximate. Verification of a high-level controller against such approximate models, although useful for finding and fixing errors quickly, cannot give us full assurance over the runtime behaviors of the controller. Therefore, we propose performing runtime monitoring of the assumptions (i.e., a discrete-state model of the environment and low-level controllers) used during design-time testing; and on detecting a divergence from the model at runtime, automatically triggering a fault recovery procedure to bring the system into a safe state. Thus, our approach combines modeling, specification, and testing with runtime monitoring and fault recovery to ensure safe execution of robotics software.

In this paper, we present Drona, a programming language framework for building safe robotics systems. We implement the software components that must satisfy critical properties using a high-level programming language called P [1]. P supports scalable compositional testing backend for analysis of the asynchronous reactive programs. Drona extends P with runtime assurance [7,8] capabilities to ensure that the assumptions made during testing about the low-level controllers and the environment hold at runtime. Drona provides features for specifying the robot workspace and also runtime libraries for deploying the generated code from P compiler on ROS [9]. The Drona toolchain is publicly available on GitHub (https://drona-org.github.io/Drona/).

1.1 Related Work

There are three popular approaches for building robotics systems with high-assurance of correctness:

(1) ***Reactive synthesis.*** There is increasing interest towards synthesizing reactive robotics controllers from temporal logic [10–13]. The programmer describes the system requirements in a high-level specification language and uses automated synthesis techniques for generating correct-by-construction high-level controllers. Tools like TuLip [14] and LTLMoP [15] construct a finite transition system that serves as an abstract model of the physical system and synthesizes a strategy, represented by a finite state automaton, satisfying the given high-level specification based on this model. Though this generated strategy is guaranteed to be safe in the abstract model of the environment, this approach has following limitations: (1) there is gap between the abstract models of the system and its actual behavior in the physical world; (2) there is gap between the generated strategy state-machine and its actual software implementation that interacts with the low-level controllers; and finally (3) the synthesis approach scale poorly both with the complexity of the mission and the size of the workspace. Other tools such as ComPlan [12] and SMC [13] generate both high-level and low-level plans, but still need additional work to translate these plans into reliable software on top of robotics platforms. Our approach is to provide a high-level language to (1) enable programmers to implement and specify the complex reactive system, (2) leverage advances in scalable systematic-testing techniques for validation of the actual implementation of the software, and (3) provide a safety envelope for operation in the real physical world via runtime assurance.

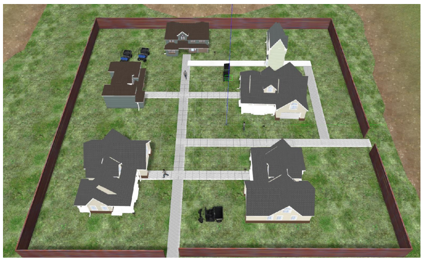
(2) ***Reachability analysis.*** Reachability analysis tools [16–18] have been used to verify robotics systems modeled as hybrid systems. If the tool successfully explores all possible reachable states of the system, then it provides a formal guarantee of correctness for the system model. Differently, from our work, reachability methods require an explicit representation of the robot dynamics and often suffer from scalability issues when the system has a large number of discrete states. Also, the analysis is performed using the models of the system, and hence, there is a gap between the models being verified and their implementation.

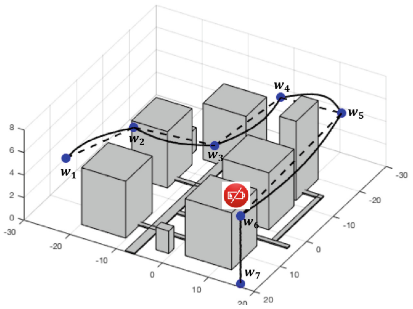
(3) ***Simulation-based falsification.*** Simulation-based tools for the falsification of CPS models (e.g., [19]) are more scalable than reachability methods, but generally, they do not provide any formal guarantees. In this approach, the entire robotics software stack is tested by simulating it in a loop with a high-fidelity model of the robot. The high-level controllers, the low-level controllers, and the robot dynamics are all executed together when exploring a behavior of the system and hence, this approach does not suffer from the gap between model and implementation described in the previous approaches. However, a challenge to achieving scalable coverage comes from the considerable time it can take for simulations.

In Drona, we decompose the validation problem into two parts: (1) we propose using systematic testing methods (known to scale for complex software systems) for high-level software and use discrete models of the low-level software (ignoring dynamics), and (2) we combine it with runtime assurance [7,8] to tackle the challenges associated with low-level controllers that involve dynamics and other uncertainties of the robotics system. Runtime verification has been applied to robotics [20–23] where online monitors are used to check the status of the system. In this paper, we use runtime assurance to address the limitations of design-time analysis.

2 Overview

We use the case study of an autonomous drone surveillance system to present our approach for programming safe robotics systems. We consider an application where a drone must autonomously patrol a set of locations in a city. Figure 1(a) shows a snapshot of the workspace in the Gazebo simulator [24]. Figure 1(b) presents the obstacle map for the workspace with the surveillance points (blue dots) and a possible path that the autonomous drone can take when performing the surveillance task (black trajectory). We consider a simplified setting where the obstacles in the workspace are always static (e.g., houses and parked cars).


(a) Workspace



(b) Obstacle Map

Fig. 1. Case study: drone surveillance system (Color figure online)

The software stack for the drone surveillance system, even in such a simplified setting, consists of multiple complex components (Fig. 2). At the top, there is the implementation of the surveillance protocol that ensures the application specific properties (e.g., repeatedly visit the surveillance points in some priority order). The rest of the components are generic components such as the motion planner, the motion primitives and the perception module that together ensure safe movement of the drone in the workspace. The surveillance protocol generates the next target location for the drone. The motion planner computes a motion plan which is a sequence of waypoints from the current location to the target location. The waypoints $w_1 \dots w_6$ in Fig. 1(b) represent one such motion plan generated by the planner and the dotted lines represent the reference trajectory for the drone.

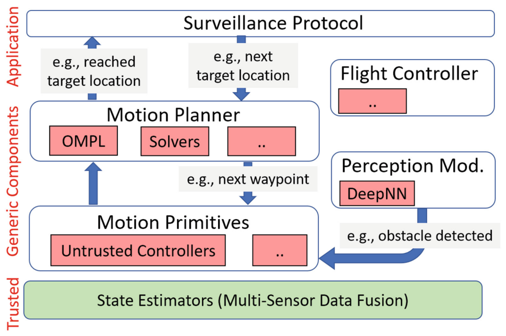


Fig. 2. Robotics software stack (Color figure online)

The motion primitives on receiving the next waypoint generate the required low-level controls necessary to follow the reference trajectory. The trajectory in Fig. 1(b) represents the actual path of the drone, which deviates from the reference trajectory because of the underlying dynamics and disturbances. The flight controller module maintains information about the mode of operation of the system and ensures that the critical events are prioritized correctly in all the modes. The perception module is used for detection obstacles and passing the information to the planner and controller to avoid a collision.

Programming the robotics software stack (Fig. 2) is challenging as it consists of multiple components, each implementing a complicated protocol, and continuously interacting with each other for accomplishing the mission safely. These components may, in turn, use third-party or machine-learning components that are hard to verify or test for correctness at design time. Hence, providing end-to-end correctness guarantees for robotics system is challenging and requires advances in both design time (static) and runtime analysis research.

2.1 Drona: Programming Framework for Safe Robotics

Figure 3 provides an overview of the Drona toolchain, a unified framework for modeling, implementing and testing safe robotics systems.

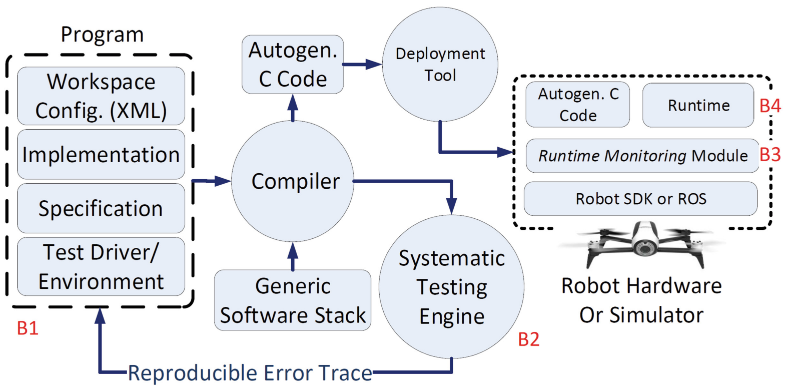


Fig. 3. Drona tool chain

The challenges in building safe robotics system span across the domains of programming languages, systematic testing, and runtime verification.

(1) ***Safe programming of reactive robotics software stack.*** For assured autonomy, the robotics software stack must be reactive to both, events happening in the environment as well as events that are triggered by a change in state of the robot. For example, the drone must correctly handle a *battery low* event, which might involve multiple components collaborating to ensure that the drone lands safely or navigates to the battery charging station in time. Safe programming of such a reactive behavior is notoriously hard as it must reliably handle concurrency and event-driven interaction among the components. Hence, the first challenge is to design a programming language that helps succinctly implement the event-driven software stack at a higher level of abstraction. Moreover, the programming language must also support modular design and implementation as the robotics software is in general built as a composition of several components.

For addressing this challenge, Drona uses the P programming language framework (Fig. 3B1). The P module system allows modular implementation of software stack where each component (protocol) is implemented as a separate module and modules are assembled (composed) together to build the complex system (Sect. 3). Further details about the P programming language is provided in Sect. 3 and in [2,3].

A Drona application consists of four blocks—*implementation*, *specification*, *workspace configuration*, and *test-driver*. The *implementation* block is a collection of P modules implementing the high-level controllers. *Specification* block capture the application specific correctness properties. The *workspace configuration* XML file provides details about the workspace, like the size of the workspace grid, location of static obstacles, location of battery charging points, and the starting position of each robot. The *test-driver* block implements the *finite* environment state machines (models) and abstractions of untrusted components to close the system for systematic testing.

(2) ***Scalable Systematic Testing of Reactive Robotics System.*** Programmers find it difficult to correctly implement an event-driven system as it involves reasoning about numerous program control paths resulting from the non-deterministic interleaving of the event handlers. Unfortunately, even the state-of-the-art systematic testing techniques scale poorly with increasing system complexity. Moreover, when implementing robotics software, the programmer may use several uncertified components (red blocks in Fig. 2). These components are hard to analyze (e.g., black box machine-learning modules) or are provided by third-party. The programming and testing framework must, therefore, provide primitives for implementing the abstractions or models of these components. Hence, the second challenge is to design a testing framework that enables scalable analysis of event-driven robotics software and also allows easy substitution of implementation modules with their abstractions (models) during testing.

One can scale systematic testing to large, industrial-scale implementations by decomposing the system-level testing problem into a collection of component-level testing problems. Moreover, the results of component-level testing can be lifted to the whole system level by leveraging the theory of assume-guarantee (AG) reasoning. Drona leverages the *compositional testing* [2,3] backend of P (Fig. 3B2) for addressing this challenge. The compiler generates a translation of the system implementation into a decomposed component-level testing problems. For each decomposed test instance the systematic testing tool enumerates executions resulting from scheduling and explicit nondeterministic choices. A programmer typically spends the initial part of development in the iterative edit-compile-test-debug loop enabled by our systematic testing tool. The feedback from the tool is an error trace that picks a particular sequence of nondeterministic choices leading to the error. Compositional testing provides orders of magnitude more test-coverage compared to the traditional monolithic systematic testing approaches [2], uncovering several critical bugs in our implementation that could have caused safety violations.

(3) *Guaranteeing safety when using untrusted components.* In practice, when building a robotics software stack, the programmer may use several uncertified components (red blocks in Fig. 2). For example, implementing an on-the-fly motion planner may involve solving an optimization problem or using an efficient search technique that relies on a solver or a third-party library (e.g., OMPL). Similarly, motion primitives are either designed using machine-learning techniques or optimized for specific tasks without considering safety or are off-the-shelf controllers provided by third parties. Ultimately, in the presence of such uncertified or hard-to-verify components, no formal guarantees can be provided using design-time testing or verification techniques. The final challenge, therefore, is to design a runtime assurance framework that guarantees the safety of the system at runtime even when the untrusted components violate the assumptions made during design time.

Drona extends the P programming framework with capabilities for runtime assurance, it supports efficient online monitoring of design time assumptions and allows the programmer to specify recovery mechanism in case the assumptions can be violated (Fig. 3B3). The idea is to wrap each untrusted component inside a runtime assurance module that monitors the assumptions made for that component during the design-time analysis and triggers a recovery procedure to guarantee that the assumptions are not violated at runtime.

The Drona compiler also generates C code that is compiled by a standard C compiler and linked against the runtime to create the executable that can be deployed on the target platform, a collection of machines, or robotics system. Runtime ensures that the behavior of a Drona program matches the semantics validated by the systematic testing. Further details about Drona toolchain and the case-study of programming distributed mobile robotics is available in [25].

3 P: Modular and Safe Event-Driven Programming

A significant challenge in building reactive robotics software is safe asynchronous event-driven programming. Asynchrony and reactivity are challenging to get right because it inevitably leads to the concurrency with its notorious pitfalls of race conditions and Heisenbugs. To address the challenges of asynchronous computation, we have developed P [1] (https://github.com/p-org/P), a (domain-specific) programming language for modeling, specifying and compositionally testing protocols in asynchronous event-driven applications.

The P programmer writes the protocol and its specification at a high-level. P provides first-class support for modeling concurrency, specifying safety and liveness properties, and checking that the program satisfies its specification [26,27]. In these capabilities, it is similar to TLA+ [28] and SPIN [29]. Unlike TLA+ and SPIN, a P program can also be compiled into executable C code. This capability bridges the gap between high-level model and low-level implementation and eliminates a massive hurdle to the acceptance of formal modeling and specification among programmers.

P got its start in Microsoft software development when it was used to ship the USB 3.0 drivers in Windows 8.1 and Windows Phone. P enabled the detection and debugging of hundreds of race conditions and Heisenbugs early on in the design of the drivers. Since then, P has been used to ship many more drivers in subsequent versions of Windows. More recently, we have used P to build fault-tolerant distributed systems [2,3] and distributed robotics systems [25].

3.1 Basic Programming Constructs

P [1] is an actor-oriented [30] programming language where actors are implemented as state machines. A P program comprises state machines communicating asynchronously with each other using events accompanied by typed data values. Each machine has an input buffer, event handlers, and a local store. The machines run concurrently, receiving and sending events, creating new machines, and updating the local store.

We introduce the key constructs of P through a simple client-server application (see Fig. 4) implemented as a collection of P state machines. In this example, the client sends a request to the server and waits for a response; on receiving a response from the server, it computes the next request to send and repeats this in a loop. The server waits for a request from the client; on receiving a request, it interacts with a helper protocol to compute the response for the client.

Events and Interfaces. An event declaration has a name and a payload type associated with it. Figure 4(a) (line 2) declares an event `eRequest` that must be accompanied by a tuple of type `RequestType`. Figure 4(a) (line 6) declares the named tuple type `RequestType`. P supports primitive types like int, bool, float, and complex types like tuples, sequences and maps. Each interface declaration has an interface name and a set of events that the interface can receive. For example, the interface `ClientIT` declared at Fig. 4(b) (line 3) is willing to receive

```
 1 /* Events */
 2 event eRequest : RequestType;
 3 event eResponse: ResponseType;
 4 ...
 5 /* Types */
 6 type RequestType =
 7 (source: ClientIT, reqId:int, val: int);
 8 type ResponseType = (resId: int, success: bool);
 9
10 machine ClientImpl receives eResponse;
11 sends eRequest; creates ServerToClientIT;
12 {
13   var server : ServerToClientIT;
14   var nextId, nextVal : int;
15   start state Init {
16     entry {
17       server = new ServerToClientIT;
18       goto StartPumpingRequests;
19     }
20   }
21   state StartPumpingRequests {
22     entry {
23       if(nextId < 5) //send 5 requests
24       {
25         send server, eRequest, (source = this,
26                 reqId = nextId, val = nextVal);
27         nextId++;
28       }
29     }
30     on eResponse do (payload: ResponseType) {
31       /* compute nextVal */
32       goto StartPumpingRequests;
33     }
34   }
35 }
```

(a) Client State Machine

```
 1 /* Interfaces */
 2 interface ServerToClientIT receives eRequest;
 3 interface ClientIT receives eResponse;
 4 interface HelperIT receives eProcessReq;
 5
 6 machine ServerImpl
 7 sends eResponse, eProcessReq;
 8 receives eRequest, eReqSuccess, eReqFail;
 9 creates HelperIT;
10 {
11   var helper: HelperIT;
12   start state Init {
13     entry {
14       helper = new HelperIT;
15       goto WaitForRequests;
16     }
17   }
18
19   state WaitForRequests {
20     on eRequest do (payload: RequestType) {
21       var client: ClientIT;
22       var result: bool;
23       client = payload.source;
24       /* interacts with the helper machine */
25       send helper, eProcessReq,
26                 (payload.reqId, payload.val);
27       ...
28       /* outcome: result = true or false*/
29       send client, eResponse,
30       (resId = payload.reqId, success = result);
31     }
32   }
33 }
34 machine HelperImpl receives eProcessReq;
35 sends eReqSuccess, eReqFail, ..; creates .. ;
36 { /* body */ }
```

(b) Server State Machine

Fig. 4. A client-server application using P state machines

only event `eResponse`. Interfaces are like symbolic names for machines. In P, unlike in the actor model where an instance of an actor is created using its name, an instance of a machine is created indirectly by performing **new** of an interface and linking the interface to the machine separately. For example, execution of the statement `server = new ServerToClientIT` at Fig. 4(a) (line 17) creates a fresh instance of machine `ServerImpl` and stores a unique reference to the new machine instance in `server`. The link between `ServerToClientIT` and `ServerImpl` is provided separately by the programmer using the `bind` operation (details in Sect. 3.2).

Machines. Figure 4(a) (line 10) declares a machine `ClientImpl` that is willing to receive event `eResponse`, guarantees to send no event other than `eRequest`, and guarantees to create (by executing `new`) no interface other than `ServerToClientIT`. The body of a state machine contains variables and states. Each state can have an entry function and a set of event handlers. The machine executes the entry function each time it enters that state. After executing the entry function, the machine tries to dequeue an event from the input buffer or blocks if the buffer is empty. Upon dequeuing an event from the input queue of the machine, the attached handler is executed. Figure 4(a) (line 30) declares an event-handler in the `StartPumpingRequests` state for the `eResponse` event, the `payload` argument stores the payload value associated with the dequeued `eResponse` event. The machine transitions from one state to another on executing the `goto` statement. Executing the statement `send t,e,v` adds event `e` with payload value `v` into the buffer of the target machine `t`. Sends are buffered, non-blocking, and directed. For example, the send statement Fig. 4(a) (line 25) sends `eRequest` event to the machine referenced by the `server` identifier. In P, the type of a machine-reference variable is the name of an interface.

Next, we walk through the implementation of the client (`ClientImpl`) and the server (`ServerImpl`) machines in Fig. 4. Let us assume that the interfaces `ServerToClientIT`, `ClientIT`, and `HelperIT` are programmatically linked to the machines `ServerImpl`, `ClientImpl`, and `HelperImpl` respectively (we explain these bindings in Sect. 3.2). A fresh instance of a `ClientImpl` machine starts in the `Init` state and executes its entry function; it first creates the interface `ServerToClientIT` that leads to the creation of an instance of the `ServerImpl` machine, and then transitions to the `StartPumpingRequests` state. In the `StartPumpingRequests` state, it sends a `eRequest` event to the server with a payload value and then blocks for a `eResponse` event. On receiving the `eResponse` event, it computes the next value to be sent to the server and transitions back to the `StartPumpingRequests` state. The `this` keyword is the "self" identifier that references the machine itself. The `ServerImpl` machine starts by creating the `HelperImpl` machine and moves to the `WaitForRequests` state. On receiving a `eResponse` event, the server interacts with the helper machine to compute the `result` that it sends back to the client.

3.2 Compositional Programming

P allows the programmer to decompose a complex system into simple components where each component is a P module. Figure 5 presents a modular implementation of the client-server application. A primitive module in P is a set of bindings from interfaces to state machines.

```
module ClientModule = {
  ClientIT -> ClientImpl
};
module ServerModule = {
  ServerToClientIT -> ServerImpl,
  HelperIT -> HelperImpl
};
//C code generation for the implementation.
implementation app: ClientModule || ServerModule;

module ServerModule' = {
  ServerToClientIT -> ServerImpl',
  HelperIT -> HelperImpl
};
implementation app': ClientModule || ServerModule';
```

Fig. 5. Modular client-server implementation

`ServerModule` is a primitive module consisting of machines `ServerImpl` and `HelperImpl` where the `ServerImpl` machine is bound to the `ServerToClientIT` interface and the `HelperImpl` machine is bound to the `HelperIT` interface. The compiler ensures that the creation of an interface leads to the creation of a machine to which it binds. For example, creation of the `ServerToClientIT` interface (executing `new ServerToClientIT`) by any machine inside the module or by any machine in the environment (i.e., outside `ServerModule`) would lead to the creation of an instance of `ServerImpl` machine.

The client-server application (Fig. 4) can be implemented modularly as two separate modules `ClientModule` and `ServerModule`; these modules can be implemented and tested in isolation. Modules in P are *open systems*, i.e., machines inside the module may create interfaces that are not bound in the module; similarly, machines may send events to or receive events from machines that are not in the module. For example, the `ClientImpl` machine in `ClientModule` creates an interface `ServerToClientIT` that is not bound to any machine in `ClientModule`, it sends `eRequest` and receives `eResponse` from machines that are not in `ClientModule`.

Composition in P (denoted ||) is supported by type checking. If the composition type checks (typing rules for module constructors are defined in [2,3]) then the composition of modules behaves like language intersection over the traces of the modules. The compiler ensures that the joint actions in the composed module (`ClientModule || ServerModule`) are linked appropriately, e.g., the creation of the interface `ServerToClientIT` (Fig. 4(a) line 18) in `ClientModule` is linked to `ServerImpl` in `ServerModule` and all the sends of `eRequest` events are enqueued in the corresponding `ServerImpl` machine. The compiler generates C code for the module in the `implementation` declaration.

Note that the indirection enabled by the use of interfaces is critical for implementing the key feature of *substitution* required for modular programming, i.e., the ability to seamlessly replace one implementation module with another. For example, `ServerModule'` (Fig. 5 line 11) represents a module where the server protocol is implemented by a different machine `ServerImpl'`. In module `ClientModule || ServerModule'`, the creation of an interface `ServerToClientIT` in the client machine is linked to machine `ServerImpl'`. The *substitution* feature is also critical for compositional reasoning, in which case, an implementation module is replaced by its abstraction.

3.3 Compositional Testing

Monolithic testing of large systems is prohibitively expensive due to an explosion of behaviors caused by concurrency and failures. The P approach to this problem is to use the principle of assume-guarantee reasoning for decomposing the monolithic system-level testing problem into component-level testing problems; testing each component in isolation using abstractions of the other components.

Spec Machines. In P, a programmer can specify temporal properties via specification machines (monitors). `spec s observes E1, E2 { .. }` declares a specification machine `s` that observes events `E1` and `E2`. If the programmer chooses to attach s to a module `M`, the code in `M` is instrumented automatically to forward any event-payload pair (e, v) to s if e is in the observes list of s; the handler for event e inside s executes synchronously with the delivery of e. The specification machines observe only the output events of a module. Thus, specification machines introduce a publish-subscribe mechanism for monitoring events to check temporal specifications while testing a P module. The module constructor `assert s in P` attaches specification machine `s` to module `P`. In Fig. 6(a),

```
machine AbstractServerImpl receives eRequest;
sends eResponse;
{
  start state Init {
    on eRequest do (payload: RequestType) {
    send payload.source, eResponse,
    (resId = payload.reqId,success = choose());
    }
  }
}
spec ReqIdsAreMonoInc observes eRequest {
  var prevId : int;
  start state Init {
    on eRequest do (payload: RequestType) {
      assert(payload.reqId == prevId + 1);
      prevId = payload.reqId;
    }
  }
}
spec ResIdsAreMonoInc observes eResponse
{
  var prevId : int;
  start state Init {
    on eResponse do (payload: ResponseType) {
      assert(payload.resId == prevId + 1);
      prevId = payload.resId;
    }
  }
}

```

(a) Abstraction and Specifications

```
module AbstractServerModule = {
  ServerToClientIT -> AbstractServerImpl
};

module AbstractClientModule = {
  ClientIT -> AbstractClientImpl
};

/* Compositional Safety Checking */
//Test: ClientModule.
test test0: (assert ReqIdsAreMonoInc in ClientModule)
              || AbstractServerModule;
//Test: ServerModule.
test test1: (assert ResIdsAreMonoInc in ServerModule)
              || AbstractClientModule;

/* Circular Assume-Guarantee */
//Check that client abstraction is correct.
test test2: ClientModule || AbstractServerModule
            refines
            AbstractClientModule || AbstractServerModule;
//Check that server abstraction is correct.
test test3: AbstractServerModule || ServerModule
            refines
            AbstractClientModule || AbstractServerModule;

// Create abstract module using Hide
module hideModule = hide X in AbstractServerModule;

test test4: ClientModule || ServerModule
            refines
            AbstractClientModule || hideModule;
```

(b) Test Declarations for Compositional Testing

Fig. 6. Compositional testing of the client-server application using P modules

`ReqIdsAreMonoInc` and `ResIdsAreMonoInc` are specification machines observing events `eRequest` and `eResponse` to assert the safety property that the `reqId` and `resId` in the payload of these events are always monotonically increasing. Note that `ReqIdsAreMonoInc` is a property of the client machine and `ResIdsAreMonoInc` is a property of the server machine.

In P, abstractions used for assume-guarantee reasoning are also implemented as modules. For example, `AbstractServerModule` is an abstraction of the `ServerModule` where the `AbstractServerImpl` machine implements an abstraction of the interaction between `ServerImpl` and `HelperImpl`. The `AbstractServerImpl` machine on receiving a request sends back a random response.

P enables decomposing the monolithic problem of checking: `(assert ReqIdsAreMonoInc, ResIdsAreMonoInc in ClientModule || ServerModule)` into four simple proof obligations. P allows the programmer to write each obligation as a test-declaration. The declaration `test tname: P` introduces a safety test obligation that the executions of module `P` do not result in a failure/error. The declaration `test tname: P refines Q` introduces a test obligation that module `P` refines module `Q`. The notion of refinement in P is trace-containment based only on externally visible actions, i.e., `P` refines `Q`, if every trace of `P` projected onto the visible actions of `Q` is also a trace of `Q`. P automatically discharges these test obligations using systematic testing. Using the theory of compositional safety [2,3], we decompose the monolithic safety checking problem into two obligations (tests) `test0` and `test1` (Fig. 6(b)). These tests use abstractions to check that each module satisfies its safety specification. Note that interfaces and the programmable bindings together enable substitution during compositional reasoning. For example, `ServerToClientIT` gets linked to `ServerImpl` in implementation but to its abstraction `AbstractServerImpl` during testing.

Meaningful testing requires that these abstractions used for decomposition be sound. To this end, P module system supports circular assume-guarantee reasoning [2,3] to validate the abstractions. Tests `test2` and `test3` perform the necessary refinement checking to ensure the soundness of the decomposition (`test0`,`test1`). The challenge addressed by our module system is to provide the theorems of compositional safety and circular assume-guarantee for a dynamic programming model of P state machines.

P module system also provides module constructors like `hide` for hiding events (interfaces) and `rename` for renaming of conflicting actions for more flexible composition. Hide operation introduces privates events (interface) into a module, it can be used to converts some of the visible actions of a module into private actions that are no longer part of its visible trace. For example, assume that modules `AbstractServerModule` and `ServerModule` use event `X` internally for completely different purposes. In that case, the refinement check between them is more likely to hold if `X` is not part of the visible trace of the abstract module. Figure 6(b) (line 28–33) show how hide can be used in such cases.

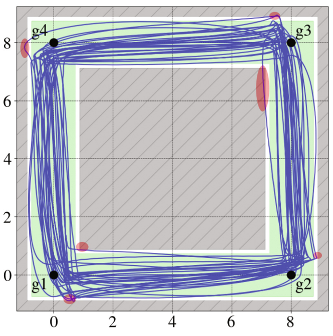
P enables modular programming of the complex robotics software stack where each component can be implemented as a separate module and compositionally tested in isolation using the principles of assume-guarantee reasoning. We have used P to implement robotics software stack for drones and found several critical bugs during the development process that could have potentially lead to a drone crash. More details along with demonstration videos are available at https://drona-org.github.io/Drona/.

4 Safe Robotics Using Runtime Assurance

When performing the compositional testing of the high-level controller, we use abstractions or models of the low-level controllers. For carrying over the analysis of high-level controllers performed at design time to runtime, one must ensure that the assumptions about the low-level controllers made during testing hold at runtime. Hence the need for a framework that supports runtime monitoring of design time assumptions and provides safety assurance if these assumptions can be violated.

Let us consider the example of guaranteeing safety in the presence of an untrusted motion primitive component. A drone navigates in the 3D space by tracking trajectories between waypoints computed by the motion planner (Fig. 2). Given the next waypoint, an appropriate motion primitive is used to track the reference trajectory. Informally, a motion primitive consists of a pre-computed control law (sequence of control actions) that regulates the state of the drone as a function of time. A motion primitive take as input the next waypoint and generates the low-level control to traverse the reference trajectory from the current position to the target waypoint. When testing the high-level controllers in the software stack, we assume that the motion primitives safely takes the drone from its current position to the target position by remaining inside the green tube and use the corresponding discrete abstraction. Since the control is optimized for performance rather than safety and also approximates the drone dynamics, it can be potentially unsafe.

To demonstrate this, we experimented with the motion primitives provided by the drone manufacturers. The drone was tasked to repeatedly visit locations g_1 to g_4 in that order,i.e., the sequence of waypoints $g_1, \dots g_4$. The blue lines represent the trajectories of the drone. Given the complex dynamics of a drone and noisy sensors, ensuring that it precisely follows a fixed trajectory (ideally a straight line joining the waypoints) is tough.



The low-level controller (untrusted third party controller) optimizes for time and, hence, during high-speed maneuvers, the reduced control on the drone leads to overshoot and trajectories that collide with obstacles (represented by the red regions). Note that the controller can be used during the majority of this mission except for a few instances of unsafe maneuvers. Hence, there is a need for runtime verification techniques that ensure the safety of the system when the design-time assumptions can be violated.

We use runtime verification to monitor at runtime the assumptions and abstractions used during systematic testing. An online monitor is useful as it can determine if any of the assumptions (specifications) can be violated and notify the operator about the unexpected behavior or trigger some correcting input actions to fix the problem. In our recent work [31], we use runtime verification to monitor the assumptions about low-level controllers and drone dynamics; we show that violations of these assumptions can be predicted in time to make corrective measure and ensure overall correctness of the mission implemented by the high-level controller. We have used runtime assurance to build end-to-end correct robotics software stack, more details about the framework and experiments is available in [32].

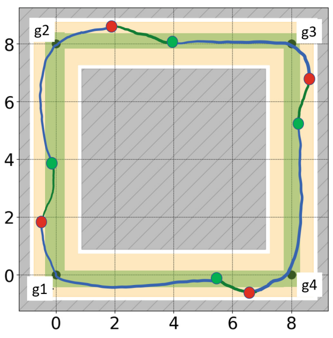


Fig. 7. Experiment using runtime assurance (Color figure online)

Let us revisit the experiment described earlier. We monitor the assumption that the drone under the influence of the motion primitive will remain inside the yellow tube.

This assumption may get violated at runtime because of various reasons, runtime assurance guarantees that it can be predicted in time and recovery operation can be triggered to bring the drone back to the safe (green) tube. Figure 7 presents one of the interesting trajectories where the recovery module takes control multiple times and ensures the overall correctness of the mission.

The red dots represent the points where the runtime monitoring system switches control to recovery, and the green dots represent the locations where it returns control to the untrusted controller for mission completion.

5 Conclusion

Drona is a novel programming framework that makes it easier to implement, specify, and compositionally test robotics software. It uses a mechanism based on runtime assurance to ensure that the assumptions about the untrusted components in the software stack hold at runtime. We firmly believe that this combination of design time techniques like programming languages and testing with runtime assurance is the right step towards solving the problem of building robust robotics systems. P enables modular implementation of the software stack and is effective in finding critical software bugs in our implementation; the runtime assurance extension helps ensure safety if there are bugs in the untrusted software components. We have evaluated Drona by deploying the generated code both on real drone platforms and running rigorous simulations in high-fidelity drone simulators. Both Drona (https://drona-org.github.io/Drona/) and P (https://github.com/p-org/P) are publicly available. For future work, we are investigating the role a system like Drona can play in the design and implementation of verified learning-based robotics, and more generally for verified artificial intelligence [33], where we believe runtime assurance will play a central role.

Acknowledgments. The work of the first and third authors was supported in part by the TerraSwarm Research Center, one of six centers supported by the STARnet phase of the Focus Center Research Program (FCRP) a Semiconductor Research Corporation program sponsored by MARCO and DARPA, by the DARPA BRASS and Assured Autonomy programs, and by Toyota under the iCyPhy center.

References

1. Desai, A., Gupta, V., Jackson, E., Qadeer, S., Rajamani, S., Zufferey, D.: P: safe asynchronous event-driven programming. In: Programming Language Design and Implementation (PLDI) (2013)
2. Desai, A., Phanishayee, A., Qadeer, S., Seshia, S.A.: Compositional programming and testing of dynamic distributed systems. Technical report UCB/EECS-2018-95, EECS Department, University of California, Berkeley, July 2018
3. Desai, A., Phanishayee, A., Qadeer, S., Seshia, S.A.: Compositional programming and testing of dynamic distributed systems. In: Proceedings of the ACM on Programming Languages (PACMPL) (OOPSLA) (2018)
4. Alur, R., Henzinger, T.A.: Reactive modules. Form. Methods Syst. Des. **15**, 7–48 (1999)
5. Abadi, M., Lamport, L.: Composing specifications. ACM Trans. Program. Lang. Syst. (TOPLAS) **15**, 73–132 (1993)
6. Abadi, M., Lamport, L.: Conjoining specifications. ACM Trans. Program. Lang. Syst. (TOPLAS) **17**, 507–535 (1995)

7. Sha, L.: Using simplicity to control complexity. IEEE Softw. **18**, 20–28 (2001)
8. Schierman, J.D., et al.: Runtime assurance framework development for highly adaptive flight control systems. Technical report AD1010277, Barron Associates, Inc., Charlottesville (2015)
9. Quigley, M., et al.: ROS: an open-source robot operating system. In: ICRA Workshop on Open Source Software (2009)
10. Kress-Gazit, H., Fainekos, G.E., Pappas, G.J.: Temporal logic based reactive mission and motion planning. IEEE Trans. Robot. **25**, 1370–1381 (2009)
11. Fainekos, G.E., Girard, A., Kress-Gazit, H., Pappas, G.J.: Temporal logic motion planning for dynamic robots. Automatica **45**, 343–352 (2009)
12. Saha, I., Ramaithitima, R., Kumar, V., Pappas, G.J., Seshia, S.A.: Automated composition of motion primitives for multi-robot systems from safe LTL specifications. In: Intelligent Robots and Systems, IROS, pp. 1525–1532. IEEE (2014)
13. Shoukry, Y., et al.: Linear temporal logic motion planning for teams of underactuated robots using satisfiability modulo convex programming. In: 56th IEEE Annual Conference on Decision and Control (CDC), pp. 1132–1137 (2017)
14. Wongpiromsarn, T., Topcu, U., Ozay, N., Xu, H., Murray, R.M.: TuLiP: a software toolbox for receding horizon temporal logic planning. In: International Conference on Hybrid Systems: Computation and Control (HSCC) (2011)
15. Finucane, C., Jing, G., Kress-Gazit, G.: LTLMoP: experimenting with language, temporal logic and robot control. In: IEEE/RSJ International Conference on Intelligent Robots and Systems (2010)
16. Frehse, G., et al.: SpaceEx: scalable verification of hybrid systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 379–395. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_30
17. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: an analyzer for non-linear hybrid systems. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 258–263. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_18
18. Duggirala, P.S., Mitra, S., Viswanathan, M., Potok, M.: C2E2: a verification tool for stateflow models. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 68–82. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_5
19. Dreossi, T., Donzé, A., Seshia, S.A.: Compositional falsification of cyber-physical systems with machine learning components. In: Barrett, C., Davies, M., Kahsai, T. (eds.) NFM 2017. LNCS, vol. 10227, pp. 357–372. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57288-8_26
20. Gat, E., Slack, M.G., Miller, D.P., Firby, R.J.: Path planning and execution monitoring for a planetary rover. In: Robotics and Automation. IEEE (1990)
21. Pettersson, O.: Execution monitoring in robotics: a survey. Robot. Auton. Syst. **53**, 73–88 (2005)
22. Lotz, A., Steck, A., Schlegel, C.: Runtime monitoring of robotics software components: increasing robustness of service robotic systems. In: International Conference on Advanced Robotics (ICAR) (2011)
23. Lee, I., Ben-Abdallah, H., Kannan, S., Kim, M., Sokolsky, O., Viswanathan, M.: A monitoring and checking framework for run-time correctness assurance (1998)
24. Koenig, N., Howard, A.: Design and use paradigms for gazebo, an open-source multi-robot simulator. In: International Conference on Intelligent Robots and Systems (IROS) (2004)

25. Desai, A., Saha, I., Yang, J., Qadeer, S., Seshia, S.A.: DRONA: a framework for safe distributed mobile robotics. In: International Conference on Cyber-Physical Systems (ICCPS) (2017)
26. Desai, A., Qadeer, S., Seshia, S.A.: Systematic testing of asynchronous reactive systems. In: Foundations of Software Engineering (FSE) (2015)
27. Mudduluru, R., Deligiannis, P., Desai, A., Lal, A., Qadeer, S.: Lasso detection using partial-state caching. In: Conference on Formal Methods in Computer-Aided Design (FMCAD) (2017)
28. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley Longman, Boston (2002)
29. Holzmann, G.: The Spin Model Checker: Primer and Reference Manual, 1st edn. Addison-Wesley Professional, Boston (2003)
30. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge (1986)
31. Desai, A., Dreossi, T., Seshia, S.A.: Combining model checking and runtime verification for safe robotics. In: Lahiri, S., Reger, G. (eds.) RV 2017. LNCS, vol. 10548, pp. 172–189. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67531-2_11
32. Desai, A., Ghosh, S., Seshia, S.A., Shankar, N., Tiwari, A.: SOTER: programming safe robotics system using runtime assurance. Technical report UCB/EECS-2018-127, EECS Department, University of California, Berkeley, August 2018
33. Seshia, S.A., Sadigh, D., Sastry, S.S.: Towards verified artificial intelligence. CoRR, vol. abs/1606.08514 (2016)