Getting Started with the P# Framework

Pantazis Deligiannis[1], Akash Lal[2], Shaz Qadeer[3]

[1] p.deligiannis@imperial.ac.uk, *Imperial College London*, UK
[2] akashl@microsoft.com, *Microsoft Research*, India
[3] qadeer@microsoft.com, *Microsoft Research*, USA

1 Introduction

P# [1][2] is an advanced systematic testing framework, designed to significantly ease the process of developing and testing asynchronous reactive applications (i.e. distributed systems and web services) in Microsoft's .NET platform. P# works as follows:

- Provides an API for writing *test harnesses*, specifying *safety* and *liveness properties*, and *modeling* the individual components of a large system as *communicating state-machines* (which is a similar concept to actors).
- During testing, the P# runtime takes control of the P# test harness (which drives the system-under-test) and all the modeled components, and systematically explores all interleavings between asynchronous events, as well as other declared sources of nondeterminism (e.g. timeouts and failures), to find bugs, such as local and global safety and liveness property violations, and runtime exceptions.
- P# can be optionally used for development of applications that will get deployed in production. P# provides a runtime where the execution of the P# state-machines is not controlled. This runtime is a thin layer built on top of TPL.

Programming model P# is built on top of the Roslyn[1] compiler and provides new C# language primitives (which are largely based on Microsoft's P [3] programming language) for creating machines, sending events from one machine to another, and writing assertions about system properties. Each machine has an input queue, states, state transitions, event handlers, fields and methods. Machines run concurrently with each other, each executing an event handling loop that dequeues an event from the input queue and handles it by executing a sequence of operations. Each operation might update a field, create a new machine, or send an event to another machine. In P#, create machine operations and send operations are non-blocking. In the case of a send operation the message is simply enqueued into the input queue of the target machine.

Usage There are many different ways that someone can use P# to test existing systems, or build new highly-reliable ones:

- P# can be used just for systematically testing an existing message-passing system, by modeling its environment (e.g. a client) and/or components of the system.
- The *surface syntax* of P# can be used to write an entire system from scratch. The surface P# syntax directly extends C# with new language constructs, which allows for rapid prototyping. However, to use the surface syntax, a developer has to use the P# compiler, which is built on top of Roslyn. The main disadvantage of this approach is that P# does not yet fully integrate with the Visual Studio integrated development environment (IDE), and thus does not support high-productivity features such as IntelliSense (e.g. for auto-completition and automated refactoring).

[1] https://github.com/dotnet/roslyn

- P# can be used as a C# library to write an entire system from scratch. This approach is slightly more verbose than the above, but allows full integration with Visual Studio. Note that most examples in this guide will use the P# surface syntax, since it is less verbose. See §4 for an example of using P# as a C# library.

Repository The P# framework is publicly available as open-source and can be found at its git repository at https://github.com/p-org/PSharp.

In the rest of this guide, we first introduce the basic features of a P# program (see §2) and then discuss how P# can be embedded in C# code (see §3). Next, we present an example of a simple program using both the P# surface syntax and P# as a library (see §4). We then present the more advanced features of P# (see §5). Next, we illustrate how someone can write safety properties in P# (see §**??**), which can be systematically checked for bugs. Finally, we provide an overview of the tools for compiling and systematically testing a P# program.

2 Basic features of a P# program

A P# program is a collection of `event` and `machine` declarations and, optionally, other top-level C# declarations, such as `class` and `struct`. All top-level declarations must be declared inside a `namespace`, as in C#. If someone uses the P# high-level syntax, then events and machines must be declared inside a `.psharp` file, while C# top-level declarations must be declared in a `.cs` file. On the other hand, if someone uses P# as a C# library, all the code must be written inside a `.cs` file.

State machines are first-class citizens of the P# language and can be declared in the following way:

```
machine Server { ... }
```

The above code snippet declares a P# machine named `Server`. Machine declarations are similar to class declarations in C#, and thus can contain an arbitrary number of fields and methods. For example, the below code snippet declares the field `client` of type `machine`. An object of this type contains a reference to a machine instance.

```
machine Server {
  machine client;
}
```

The main difference between a class and a machine declaration is that the latter must also declare one or more *states*:

```
machine Server {
  machine client;
  start state Init { ... }
  state Active { ... }
}
```

The above declares two states in the `Server` machine: `Init` and `Active`. The P# developer must use the `start` modifier to declare an *initial* state, which will be the first state that the machine will transition to upon instantiation. In this example, the `Init` state has been declared as the initial state of `Server`. Note that only a single state is allowed to be declared as an initial per machine. A `state` declaration can optionally contain a number of state-specific actions, as seen in the following code snippet:

```
state SomeState {
  entry { ... }
  exit { ... }
}
```

A code block indicated by `entry` { ... } denotes an action that will be executed when the machine transitions to the state, while a code block indicated by `exit` { ... } denotes an action that will be executed when the machine leaves the state. Actions in P# are essentially C# methods with no input parameters and `void` return type. P# actions can contain arbitrary P# and C# statements. However, since we want to explicitly declare all sources of asynchrony using P#, we only allow the use of *sequential* C# code inside a P# machine.[2] An example of an `entry` action is the following:

```
entry {
  this.client = create(Client);
  send(this.client, Config, this);
  send(this.client, Ping);
  raise(Unit);
}
```

The above action contains the three most important P# statements. The `create` statement is used to create a new instance of the `Client` machine. A reference to this instance is stored in the `client` field. Next, the `send` statement is used to send an event (in this case the events `Config` and `Ping`) to a target machine (in this case the machine whose address is stored in the field `client`).

When an event is being send, it is enqueued in the event queue of the target machine, which can then dequeue the received event, and handle it asynchronously from the sender machine. Finally, the `raise` statement is used to send an event to the caller machine (i.e. to itself). When a machine raises an event, the raised event is not enqueued as in the case of `send`; instead, the machine terminates execution of the enclosing code block and handles the event immediately. In P#, events (e.g. `Ping`, `Unit` and `Config` in the above example) can be declared as follows:

```
event Ping;
event Unit;
event Config (target: machine);
```

A P# machine can send data (scalar values or references) to a target machine, as the payload of an event. Such an event must specify the type of the payload in its declaration (as in the case of the `Config` event above). A machine can also send data to itself (e.g. for processing in a later state) using `raise`.

In the previous example, the `Server` machine sends `this` (i.e. a reference to the current machine instance) to the `client` machine. The receiver (in our case `client`) can retrieve the sent data by using the keyword `trigger` (or `ReceivedEvent` when using P# as a C# library), which is a handle to the received event, casting `trigger` to the expected event type (in this case `Config`), and then accessing the payload as a field of the received event.

As discussed earlier, the `create` and `send` statements are non-blocking. The P# runtime will take care of all the underlying asynchrony using the Task Parallel Library and, thus, the developer does not need to explicitly create and manage tasks.

Besides the `entry` and `exit` declarations, all other declarations inside a P# state are related to *event-handling*, which is a key feature of P#. An event-handler declares how a P# machine should *react* to a received event. One such possible reaction is to create one or more machine instances, send one or more events, or process some local data. The two most important event-handling declarations in P# are the following:

```
state SomeState {
  on Unit goto AnotherState;
  on Pong do SomeAction;
}
```

The declaration `on Unit goto AnotherState` indicates that when the machine receives the `Unit` event in `SomeState`, it must handle the event by exiting the state and transitioning to `AnotherState`.

[2] In practise, we just assume that the C# code is sequential, as it would be very challenging to impose this rule in real life programs (e.g. a developer could use an external library).

The declaration `on Pong do SomeAction` indicates that the `Pong` event must be handled by invoking the action `SomeAction`. Each event can be associated with at most one handler in a particular state of a machine. P# also supports *anonymous* event-handlers. For example, the declaration `on Pong do` { ... } is an anonymous event-handler, which states that the block of statements between the braces must be executed when event `Pong` is dequeued.

P# also supports specifying invariants (i.e. assertions) on the local state of a machine. The developer can achieve this by using the `assert` statement, which accepts as input a predicate that must always hold in that specific program point, e.g. `assert(k == 0)`, which holds if the integer `k` equals to 0.

3 Interoperability Between P# and C#

Because P# is built on top of the C# language, the entry point of a P# program (i.e. the first machine that the P# runtime will instantiate and execute) must be explicitly declared inside a host C# program (typically in the `Main` method), as follows:

```
using Microsoft.PSharp;
public class HostProgram {
  static void Main(string[] args) {
    PSharpRuntime.Create().CreateMachine(typeof(Server));
    Console.ReadLine();
  }
}
```

The developer must first import the P# runtime library (`Microsoft.PSharp.dll`), then create a `PSharpRuntime` instance, and finally invoke the `CreateMachine` runtime method to instantiate the first P# machine (`Server` in the above example).

The `CreateMachine` method is part of the .NET interoperability API (a set of methods for calling P# from native C# code) that is exposed by `PSharpRuntime`. This method accepts as a parameter the type of the machine to be instantiated, and returns an object of the `MachineId` type, which contains a reference to the created P# machine. Because `CreateMachine` is an asynchronous method, we call the `Console.ReadLine` method, which pauses the main thread until a console input has been given, so that the host C# program does not exit prematurely.

The `PSharpRuntime` .NET interoperability API also provides the `SendEvent` method for sending events to a P# machine from native C#. This method accepts as parameters an object of type `MachineId`, an event and an optional payload. Although the developer has to use `CreateMachine` and `SendEvent` to call P# code from native C#, the opposite is straightforward, as it only requires accessing a C# object from P# code.

4 A Simple Example Program

The following P# program shows a `Client` machine and a `Server` machine that communicate asynchronously by exchanging `Ping` and `Pong` events:

```
namespace PingPong {
  event Ping; // Client sends this event to the Server
  event Pong; // Server sends this event to the Client
  event Unit; // Event used for local transitions

  // Event used for configuration, can take a payload
  event Config (target: machine);

  machine Server {
    machine client;
```

```

    start state Init {
      entry {
        // Instantiates the Client
        this.client = create(Client);
        // Sends event to client to configure it
        send(this.client, Config, this);
        raise(Unit); // Sends an event to itself
      }

      on Unit goto Active; // Performs a state transition
    }

    state Active {
      on Ping do {
        // Sends a Pong event to the Client
        send(this.client, Pong);
      };
    }
  }

  machine Client {
    machine server;

    start state Init {
      on Config do Configure; // Handles the event
      on Unit goto Active; // Performs a state transition
    }

    void Configure() {
      // Receives reference to Server
      this.server = (trigger as Config).target;
      raise(Unit); // Sends an event to itself
    }

    state Active {
      entry {
        SendPing();
      }
      on Pong do SendPing;
    }

    void SendPing() {
      // Sends a Ping event to the Server
      send(this.server, Ping);
    }
  }

  public class HostProgram {
    static void Main(string[] args) {
      PSharpRuntime.Create().CreateMachine(typeof(Server));
      Console.ReadLine();
    }
  }
}
```

In the above example, the program starts by creating an instance of the `Server` machine (line 61). The implicit constructor of each P# machine initializes the internal to the P# runtime data of the machine, including the event queue, a set of available states, and a map from events to event-handlers per state.

After the `Server` machine has initialized, the P# runtime executes the `entry` action of the initial (`Init`) state of `Server`, which first creates an instance of the `Client` machine (line 15), then sends the event `Config` to the `Client` machine (line 17), with the `this` reference as a payload, and then raises the event `Unit` (line 18). As mentioned earlier, when a machine calls `raise`, it exits the currently executing action, and immediately handles the raised event (bypassing the queue). In this case, the `Server` machine handles `Unit` by transitioning to the `Active` state (line 21).

`Client` starts executing (asynchronously) when it is created by `Server`. The `Client` machine stores the received payload (which is a reference to the `Server` machine) in the `server` field (line 42), and then raises `Unit` to transition to the `Active` state. In the new state, `Client` calls the `SendPing` method to send a `Ping` event to `Server` (line 55). In turn, the `Server` machine dequeues `Ping` and handles it by sending a `Pong` event to `Client` (line 27), which subsequently responds by sending a new `Ping` event to `Server`. This asynchronous exchange of `Ping` and `Pong` events continues indefinitely.

The above example can be written using P# as a C# library as follows:

```
// PingPong.cs
using System;
using Microsoft.PSharp;

namespace PingPong {
  class Unit : Event { }
  class Ping : Event { }
  class Pong : Event { }

  class Config : Event {
    public MachineId Target;
    public Config(MachineId target) : base() {
      this.Target = target;
    }
  }

  class Server : Machine {
    MachineId Client;

    [Start]
    [OnEntry(nameof(InitOnEntry))]
    [OnEventGotoState(typeof(Unit), typeof(Active))]
    class Init : MachineState { }

    void InitOnEntry() {
      this.Client = this.CreateMachine(typeof(Client));
      this.Send(this.Client, new Config(this));
      this.Raise(new Unit());
    }

    [OnEventDoAction(typeof(Pong), nameof(SendPing))]
    class Active : MachineState {
      protected override void OnEntry() {
        (this.Machine as Server).SendPing();
      }
    }

```

```
    void SendPing() {
      this.Send(this.Client, new Ping());
    }
  }

  class Client : Machine {
    MachineId Server;

    [Start]
    [OnEventGotoState(typeof(Unit), typeof(Active))]
    [OnEventDoAction(typeof(Config), nameof(Configure))]
    class Init : MachineState { }

    void Configure() {
      this.Server = (this.ReceivedEvent as Config).Trigger;
      this.Raise(new Unit());
    }

    [OnEventDoAction(typeof(Ping), nameof(SendPong))]
    class Active : MachineState { }

    void SendPong() {
      this.Send(this.Server, new Pong());
    }
  }

  public class Program {
    static void Main(string[] args) {
      PSharpRuntime.CreateMachine(typeof(Server));
      Console.ReadLine();
    }
  }
}
```

The programmer can use P# as a library by importing the `Microsoft.PSharp.dll` library. A P# machine can be declared by creating a C# `class` that inherits from the type `Machine` (provided by the P# library). A state can be declared by creating a `class` that inherits from the type `MachineState`. This state class must be nested inside a machine class (no other class besides a state can be nested inside a machine class). The start state can be declared using the `[Start]` attribute.

A state transition can be declared using the `[OnEventGotoState(...)]` attribute, where the first argument of the attribute is the type of the received event and the second argument is the type of the target state. An optional third argument, is a string that denotes the name of the method to be executed after exiting the state and before entering the new state. Likewise, an action handler can be declared using the `[OnEventDoAction(...)]` attribute, where the first argument of the attribute is the type of the received event and the second argument is the name of the action to be executed. All P# statements (e.g. `send` and `raise`) are exposed as method calls of the `Machine` and `MachineState` classes.

Note that even when using P# as a C# library, the program has to still be compiled using the P# compiler as the compiler performs some important static checking to find P# syntax-related errors and rewriting (e.g. a `return` statement is instrumented after a `raise`).

4.1 Compiling and Testing P# Programs

To compile a P# program, the developer must use the P# compiler (`PSharpCompiler.exe`), which is built on top of the Microsoft Roslyn compiler. The P# compilation process consists of two phases:

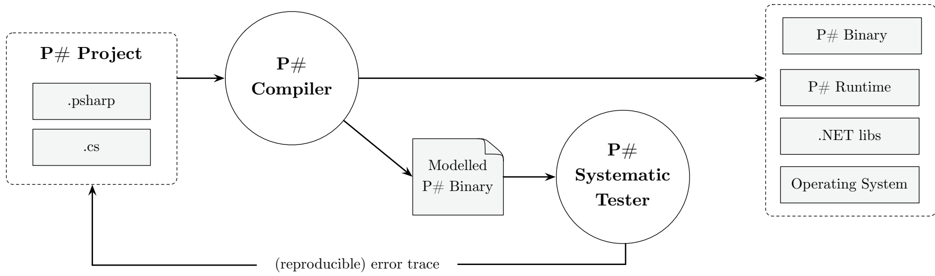

Fig. 1. The typical P# workflow.

parsing and *rewriting*. In the parsing phase, the input P# program is parsed using a recursive-descent parser to produce an abstract syntax tree (AST). In the rewriting phase, the P# compiler traverses the produced AST, and rewrites all P# statements to native (intermediate) C# code. Finally, the P# compiler invokes the Roslyn compiler to build the intermediate C# program, link it with the P# runtime library, and produce a .NET executable.

To test a P# program, the developer must use the `PSharpTester.exe` systematic testing tool (see https://github.com/p-org/PSharp for instructions), or use the P# systematic testing APIs, as follows:

```
using System;
using System.Collections.Generic;

using Microsoft.PSharp;
using Microsoft.PSharp.SystematicTesting;
using Microsoft.PSharp.Utilities;

namespace Example
{
   public class Test
   {
      static void Main(string[] args)
      {
         var configuration = Configuration.Create().
            WithLivenessCheckingEnabled().
            WithNumberOfIterations(10).
            WithVerbosityEnabled(2);
         TestingEngine.Create(configuration, Execute).Run();
      }

      [Microsoft.PSharp.Test]
      public static void Execute(PSharpRuntime runtime)
      {
         runtime.CreateMachine(typeof(SomeMachine));
      }
   }
}
```

The developer must first create a `Configuration` object, which declares testing options such as the number of testing iterations. Next, the developer must create a `TestingEngine`, passing the configuration instance and the entry point to the test (in our case `Execute`), which must be annotated with the `[Microsoft.PSharp.Test]` attribute, as arguments. Finally, the `Run` method must be invoked to start testing the P# program.

5 Advanced features of P#

The following is a discussion of more advanced features of P#, such as termination of machines, specifying safety and liveness properties, and modeling the environment.

5.1 Termination of P# machines

In order to terminate a P# machine cleanly, it must dequeue a special event named `halt`, which is provided by P# (the user cannot declare it). A `halt` event (`Halt` when using P# as a library) can be raised and/or send to another machine. Termination of a machine due to an unhandled `halt` event is valid behavior (the P# runtime does not report an error). From the point of view of formal operational semantics, a halted machine is fully receptive and consumes any event that is sent to it. The P# runtime implements this semantics efficiently by cleaning up resources allocated to a halted machine and recording that the machine has halted. An event sent to a halted machine is simply dropped. A halted machine cannot be restarted; it remains halted forever.

5.2 Writing safety properties

The most important safety specification of a P# program is that every event dequeued by a machine is handled; otherwise, the P# runtime reports an unhandled event error. The PingPong program satisfies this specification since the `Server` machine handles the `Ping` event and the `Client` machine handles the `Pong` event in every state where an event dequeue is possible.

TODO

5.3 Writing liveness properties

TODO

5.4 Modeling the environment

The following example illustrates the more advanced features of the P# language. This program implements a failure detection protocol. A `FailureDetector` machine is given a list of machines, each of which represents a daemon running at a node of a distributed system. The `FailureDetector` sends each machine in the list a `Ping` event and determines whether a machine has failed. A machine has failed if it does not respond with a `Pong` event within a certain time period. The `FailureDetector` uses an operating system timer to implement the bounded wait for the `Pong` event.

TODO

References

1. Pantazis Deligiannis, Alastair F. Donaldson, Jeroen Ketema, Akash Lal, and Paul Thomson. Asynchronous programming, analysis and testing with state machines. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 154–164. ACM, 2015.
2. Pantazis Deligiannis, Matt McCutchen, Paul Thomson, Shuo Chen, Alastair F. Donaldson, John Erickson, Cheng Huang, Akash Lal, Rashmi Mudduluru, Shaz Qadeer, and Wolfram Schulte. Uncovering bugs in distributed storage systems during testing (not in production!). In *Proceedings of the 14th USENIX Conference on File and Storage Technologies*, pages 249–262. Usenix, 2016.
3. Ankush Desai, Vivek Gupta, Ethan K. Jackson, Shaz Qadeer, Sriram K. Rajamani, and Damien Zufferey. P: Safe asynchronous event-driven programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 321–332. ACM, 2013.