Getting Started with P#

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] is an extension of the C# language, designed to significantly ease the process of developing and testing asynchronous applications in Microsoft's .NET platform. To achieve this goal, P# has the following core capabilities:

- Exposes a computational model based on *communicating state-machines* (similar concept to actors from other asynchronous programming languages). P# machines are built on top of the Task Parallel Library (TPL) and communicate by explicitly *sending* and implicitly *receiving events*, an approach commonly used in building web services and distributed systems. Because P# *fully interoperates* with C#, the P# programmer can easily blend P# and C# code, which makes it *easy to integrate* with existing .NET projects.
- Provides powerful mechanisms for writing *concurrency unit tests* of P# applications, as well as legacy C# code that uses message-passing. When executing a concurrency unit test, the P# runtime is invoked in *systematic testing* mode, where it takes control of the underlying task scheduler and systematically explores *event handler interleavings* to find bugs, such as safety and liveness property violations, and runtime exceptions.
- Enables the developer to specify *models* of the application's *environment* (e.g. the client, a sub-module of a distributed system or the network). These models can introduce non-deterministic choices (e.g. to model node failures) that are captured by the P# runtime and/or abstract away implementation details of the real system that are irrelevant to the properties being checked. When systematically testing a P# application, these models *substitute* the real environment. This can be effectively used for *compositional* (and thus scalable) testing of large systems.

Programming model P# is built on top of the Roslyn[1] compiler and provides new language primitives (which are largely based on Microsoft's P [2] 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

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

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 create highly-reliable asynchronous .NET applications:

- Use the *surface syntax* of P# to write an entire application from scratch. The surface syntax directly extends C# with new language constructs, which allows for rapid prototyping. 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).
- Use P# as a C# library to write an entire application from scratch. This approach is 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 as it is less verbose. See Section **??** for an example of using P# as a C# library.
- Mix P# projects with C# projects. P# compiles to C# using Roslyn, which allows easy integration of the two languages in the same solution.
- Use P# only for modeling the environment of a legacy C# system and then systematically test it. Note that the assumption is that the C# code that will be tested is sequential and deterministic. If not, bugs might still be found, but the P# runtime cannot fully control the underlying scheduler and, thus, loses efficiency.

Repository The P# language 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 basic features of a P# program (Section 2) and then discuss how P# can be embedded in C# code (Section 3). Next, we present an example of a simple program using both the P# surface syntax and P# as a library (Section 4). We then present the more advanced features of P# (Section 5). Next, we illustrate how someone can write safety properties in P# (see Section **??**), which can be systematically checked for bugs. Finally, we provide an overview of the tools for compiling and systematically testing a P# program and conclude with a glossary of all P#-specific features in our language.

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 { ... }
```

This declares a machine named `Server`. Machine declarations are very similar to class declarations in C# and can contain fields, states and methods. Fields can be declared similar to how fields are declared in C#:

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

This declares a field named `client` of type **`machine`**, which contains the *unique id* of a dynamically-created P# machine. This id can be used to send an event to `client`, as discussed later in the guide. Fields in P# can have an arbitrary type; all C# types, as well as user-defined types, are allowed. Note that the fields of a machine *cannot* be **`public`** or **`internal`**; they can only be accessed from the local scope of an instance of a machine. If a user declares a field as **`public`** or **`internal`**, then the compiler will exit with an appropriate error.

Machine states can be declared as follows:

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

This declares two states in the `Server` machine: `Init` and `Active`. A machine *must* declare an *initial* state, which will be the first state that the machine will start execution. The **`start`** modifier is used for declaring an initial state. In the above example, the `Init` state has been declared as the initial state of the `Server` machine.

A **`state`** declaration can optionally contain a number of state-specific declarations, as seen in the following example:

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

A code block indicated by **`entry`** { ... } denotes an action that is executed when the state is entered, while a code block indicated by **`exit`** { ... } denotes an action that is executed when the state exits. The **`entry`** and **`exit`** actions are blocks of arbitrary P# and C# statements. An example of an **`entry`** action is the following:

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

The above action contains three of the most important P# statements. The **`create`** statement, creates a new instance of the `Client` machine and stores the unique id of this instance in the field `client`. Next, the **`send`** statement sends an event (in this case the event `Ping`) to a target machine (in this case the machine whose address is

stored in the field `Client`). When an event is sent, it is enqueued in the event queue of the target machine. The target machine can then dequeue the event and handle it concurrently from the sender machine. Finally, The **`raise`** statement sends an event to the caller machine (i.e. sends an event to itself). In P#, when a machine raises an event, the raised event is not enqueued; the machine terminates execution of the enclosing code block and handles the event immediately.

Events (e.g. `Ping` and `Unit` in our case) can be simply declared as follows:

```
event Ping;
event Unit;
```

A machine can optionally send data to another machine, either when creating a new machine (via the **`create`** statement) or when sending an event to a machine (via the **`send`** statement). A machine can also send data to itself (e.g. for processing in a later state) via the **`raise`** statement. In the previous example, the machine sends **`this`** to `client`. By default, **`this`** denotes a reference to the instance of the machine that uses it, but when sending **`this`** to another machine, it sends the unique id of the machine (e.g. so the other machine can use it to send something back). The receiver machine (in our case `client` can retrieve the sent data by using the keyword **`payload`** and casting it to its expected type; in this case the **`payload`** has to be casted to the **`machine`** type (as it is the id of the sender machine).

As discussed earlier, the **`create`** and **`send`** statements are non-blocking. The P# runtime will take care of all the underlying asynchrony, and thus the user does not need to create and manage explicit tasks. For efficiency, the P# runtime executes the event handler on the context of a separate TPL task instead of a more heavyweight thread.

Other than the **`entry`** and **`exit`** declarations, all other declarations inside a state are related to *event-handling*. Two of the 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`. 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.

Actions in P# are methods with no parameters. A P# method (declared exactly like a typical C# method) can contain arbitrary P# and C# statements. Note that the methods of a machine *cannot* be **`public`** or **`internal`** (similar to machine fields). If a user declares a method as **`public`** or **`internal`**, then the compiler will exit with an appropriate error. P# also supports *anonymous* actions. For example, the declaration **`on`** `Pong` **`do`** { ... } is such an anonymous action, which states that the block of statements between the braces must be executed when event `Pong` is dequeued.

3 Embedding P# into C#

Interoperability between the two languages is in the heart of the P# programming approach. This is because P# is essentially C#, but with some extra key features for enabling safer and easier-to-test asynchronous .NET programming. Even to start executing a P# program, the programmer has to create the very first machine from inside the context of a C# method. This can be done by calling the `CreateMachine` method (provided by the `PSharpRuntime` APIs) from any C# method (typically the `Main` method) as follows:

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

The `CreateMachine` method accepts as an argument the type of the machine to be created (in our case `Server`) and an optional payload (not used in this example). Because `CreateMachine` is non-blocking, we use the `Console.ReadLine()` statement so that the program does not exit prematurely.

The `PSharpRuntime` APIs also provide the `SendEvent` method for enqueueing events to a P# machine from C# code (also not used in this example). This method accepts as an argument a `MachineId` object (which corresponds to the unique id of a machine), an event and an optional payload. The `MachineId` object is returned when creating a machine via the `CreateMachine` method.

Although the programmer has to use the above methods to send data from C# code to a P# machine, the opposite is much more straightforward. A typical way to achieve this is to pass a reference to a C# object as a payload to a P# machine, and then invoke a method (or access a field) on that object.

4 A simple P# program

The following example contains a `Server` machine and a `Client` machine communicating with each other via sending and receiving `Ping` and `Pong` events.

```
// PingPong.psharp
namespace PingPong {
  event Ping;
  event Pong;
  event Unit;

  machine Server {
    machine client;

    start state Init {
      entry {
        this.client = create(Client, this);
        raise(Unit);
      }
      on Unit goto Active;
```

```
    }

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

    void SendPing() {
      send(this.client, Ping);
    }
  }

  machine Client {
    machine server;

    start state Init {
      entry {
        this.server = (machine)payload;
        raise(Unit);
      }
      on Unit goto Active;
    }

    state Active {
      on Ping do {
        send(this.server, Pong);
      };
    }
  }

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

Listing 1.1. Simple PingPong application written in the P# high-level syntax.

In this example, the P# program starts with a single instance of `Server` being created and then entering state `Init`. Let us call this instance of the Server machine *server*. Machine *server* then creates an instance of the `Client` machine and raises the event `Unit` to enter the state `Active`. Let us call this instance of the `Client` machine *client*. The machine *client* begins execution when *server* is in the state `Active`, then *client* also transitions to the state `Active` by raising a event `Unit`. From this point on, *server* and *client* keep exchanging `Ping` and `Pong` events (in an infinite loop).

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 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);
      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))]
    class Init : MachineState {
      protected override void OnEntry() {
        (this.Machine as Client).Server = (MachineId)this.Payload;
        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();
    }
  }
}
```

Listing 1.2. Simple PingPong application written using P# as a C# library.

The programmer can use P# as a library by importing the `Microsoft.PSharp` 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.

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 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. Moreover a P# program can only be systematically tested via a special mode of the P# compiler as discussed in Section **??**.

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 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.

5.3 Writing liveness properties

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.

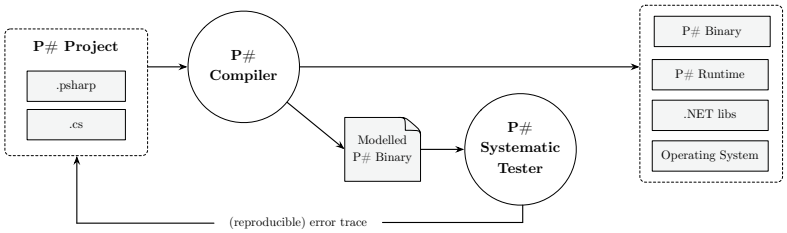

Fig. 1. The typical P# workflow.

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, 2015.
2. Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram 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, 2013.