The P# Tutorial

Pantazis Deligiannis
p.deligiannis@imperial.ac.uk

Shaz Qadeer
qadeer@microsoft.com

Akash Lal
akashl@microsoft.com

1. Introduction

P# is an actor-based programming language for develoPing highly-reliable event-driven asynchronous .NET applications. The computational model underlying a P# program is *communicating state-machines* (similar concept to actors). P# machines communicate by sending and receiving *events*, an approach commonly used in building embedded, networked, and distributed systems. P# is an extension of the C# language: it provides new language primitives for creating machines and sending events from a machine to another. Because P# is built on top of C#, the programmer can blend P# and C# code: this not only lowers the overhead of learning a new language, but also allows P# to easily integrate with existing .NET projects.

In P#, machines are first class citizens of the language (similar to how classes are first class citizens in C#). Each machine has an input queue, states, 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#, a send operation is non-blocking; the message is simply enqueued into the input queue of the target machine.

In the rest of this tutorial, we will introduce the features of P# through a series of examples. We will first present a simple program (Section 2) that illustrates the basic features of the P# state-machine programming model. Next, we present a more advanced example (Section 3) that illustrates advanced features of the P# language. Finally, we will provide an overview of the tools for compiling and systematically testing a P# program (Section 4) and conclude with a glossary of all P#-specific features in our language (Section 5).

2. A simple P# program

A P# program is a collection of `event`, `machine` and, optionally, other top-level C# declarations (e.g. `class`). Events and machines must be declared inside a `.psharp` file, while C# top-level declarations must be declared in a typical `.cs` file. Similar to C#, all top-level declarations must be declared inside a `namespace`. The following example contains a `Client` machine and a `Server` machine communicating with each other via `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 Playing;
    }

    state Playing {
      entry {
        send this.client, Pong;
      }

      on Unit do SendPong;
      on Ping do SendPong;
    }

    void SendPong() {
      send this.client, Pong;
    }
  }

  machine Client {
    machine server;
    int counter;

    start state Init {
      entry {
        this.server = (machine)payload;
        this.counter = 0;
        raise Unit;
      }

      on Unit goto Playing;
    }

    state Playing {
      entry {
        if (this.counter = 5) {
          send this.server, halt;
          raise halt;
        }
      }

      on Unit goto Playing;

      on Pong do {
        this.counter++;
        send this.server, Ping;
        raise Unit;
      };
    }
  }

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

A `machine` declaration contains a collection of field, state and method declarations. For example, the machine `Server` has a `client` field, two states, `Init` and `Playing`, and a method `SendPong` declared inside it. The fields and methods of a machine *cannot* be public; they can only be accessed locally from a particular instance of the machine. The `start` modifier in the `Init` state declaration indicates that an instance of `Server` begins executing by entering the `Init` state.

A `state` declaration can optionally contain a number of state-specific declarations. A code block indicated by `entry` { ... } denotes an action that is executed when the state is entered, while a code block indicated by `exit` { ... } (not used in the above example) denotes an action that is executed when the state exits. A P# action (e.g. `entry` `exit` and `SendPong`), is a block of arbitrary P# and C# statements. As an example, the `entry` action in the state `Init` of `Server` contains two statements. The first statement, `this.client = create Client ( this )`, creates a new instance of the `Client` machine and stores a reference to this instance in the field `client`. Note that P# allows the use of all the primitive types that C# provides (e.g. `int` and `bool`) and also introduces the `machine` type, which is a reference to a dynamically-created P# machine. The second statement in the `entry` action *raises* an event `Unit`, which causes control to exit the `Init` state and enter the `Playing` state. When a machine raises an event, it literally sends an event to itself. In P#, when a machine raises an event, or sends an event to another machine, the event is enqueued in the target machine's input queue. However, a raised event does not go through the queue; rather it terminates execution of the enclosing code block and is handled immediately.

Other than the `entry` and `exit` declarations, all other declarations inside a state are related to event handling. The declaration `on Unit goto Playing` in the state `Init` of the `Server` machine is an example of such a declaration: it indicates that the `Unit` event must be handled by exiting the state `Init` and transitioning to the state `Playing`. The declaration `on Ping do SendPong` in the state `Init` of the `Server` machine is another example: it indicates that the `Ping` event must be handled by invoking the action `SendPong`. P# also supports *anonymous* actions: the declaration `on Pong do` { ... } in the state `Playing` of the `Client` machine is such an anonymous action, which states that the code block between the braces must be executed when `Pong` is dequeued. When an action executes control remains in the state subsequent to this execution. This control primitive is useful for executing an event-driven loop in a state, such as the one in `Playing` for collecting `Pong` responses. Different states of a machine can choose to handle a particular event in a different way: the programmer is free to decide how an event should be handled.

A P# action can be any C# method that has no parameters. An example of an action is `SendPong`, which contains the `send` statement that is responsible for sending an event (in this case `Pong`) to the machine whose address is stored in the field `client`. The keyword `this` refers to the current instance of the machine (in this case an instance of the `Server` machine).

A machine can optionally send data to another machine, either when creating a machine (via the `create` statement) or when sending an event to a machine (via the `send` statement). Note that both `create` and `send` statements are non-blocking. In the above example, the `Server` machine sends a reference to itself (i.e. `this`) when creating the `Client` machine in the `entry` action of the `Init` state. The `Client` machine can retrieve this data by using the keyword `payload`, as in the `entry` action of the state `Init`. When a machine receives a `payload` it has to cast the data to its expected type; in this case the `machine` type (as it is a reference to the `Server` machine).

To start execution of a P# program, the programmer has to create an initial machine using the `CreateMachine` method provided by the `PSharpRuntime` API. This method accepts as a parameter the type of the machine to be created (in our case a `Server` machine) and an optional payload (not used in the above example). Because the `CreateMachine` method is non-blocking, we use the `Console.ReadLine()` statement so that the program does not exit prematurely. The `PSharpRuntime` API also provides a method for enqueueing events to a P# machine from C# code (not used in the above example). To do this, the programmer has to use the `SendEvent` method that accepts as a parameter a `MachineId` object, returned when creating a machine via the `CreateMachine` method, and an event object.

In the above example, the P# program begins 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 `Playing`. Let us call this instance of the `Client` machine *client*. The machine *client* begins execution when *server* is in the state `Playing`, then *client* also transitions to the state `Playing` by raising a event `Unit`. From this point on, *client* and *server* exchange `Ping` and `Pong` events.

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.

In order to terminate a P# machine cleanly, it must dequeue a special `halt` event. In this example, after *client* sends 5 `Ping` events to *server*, it sends `halt` to *server* and then raises `halt`. 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.

3. Advanced P# features

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.

3.1. Modeling a machine

The code snippet below shows the implementation of the Timer machine. This machine is declared using the keyword model (instead of machine) to indicate that it represents an abstraction of the operating system (OS) timer. This abstraction *substitutes* the OS timer and is used to *systematically test* the interaction of the failure detector protocol with the underlying OS (see Section 4). During execution of a P# program, model machines are *replaced* by their actual implementations; a model machine is used only for systematic testing.

```
// Timer.psharp
namespace FailureDetector {
  // events from client to timer
  event Start;
  event Cancel;
  // events from timer to client
  event Timeout;
  event CancelSuccess;
  event CancelFailure;
  // local event for control transfer within timer
  event Unit;

  model Timer {
    machine client;

    start state Init {
      entry {
        this.client = (machine)payload;
        raise Unit; // goto handler of Unit
      }

      on Unit goto WaitForReq;
    }

    state WaitForReq {
      on Cancel goto WaitForReq with {
        send this.client, CancelFailure ( this );
      };

      on Start goto WaitForCancel;
    }

    state WaitForCancel {
      ignore Start;

      on Cancel goto WaitForReq with {
        if ($) {
          send this.client, CancelSuccess ( this );
        } else {
          send this.client, CancelFailure ( this );
          send this.client, Timeout ( this );
        }
      };

      on default goto WaitForReq with {
          send this.client, Timeout ( this );
      };
    }
  }
}
```

Each instance of a Timer machine has a reference to a Client machine instance, stored in the client field. This client sends Start and Cancel events to the timer, which subsequently responds with Timeout, CancelSuccess, or CancelFailure events.

The timer has three states: Init, WaitForReq and WaitForCancel. It starts executing in the state Init, where it initializes the client field with the machine reference obtained by accessing payload and casting it to the type machine. It then waits in the state WaitForReq for a request from the client, responding with CancelFailure to a Cancel event and moving to the WaitForCancel state on a Start event.

The ignore Start declaration inside the WaitForCancel state of the timer denotes that whenever the event Start is dequeued it must be dropped without taking any further action.

The response to a Cancel event involves the use of nondeterminism to model the race condition between the arrival of a Cancel event from the client and the elapse of the timer. This nondeterminism is indicated by an if statement guarded by the keyword $, which returns a nondeterminitic boolean choice. The then-branch models the case when the Cancel event arrives before the timer elapses; in this case, CancelSuccess is sent back to the client. The else-branch models the case when the timer fires before the Cancel event arrives; in this case, CancelFailure and Timeout are sent back to the client one after another. The last event handler `on default goto WaitForReq with { ... }` transfers the control to the WaitForReq state, modeling that the timer has elapsed, if neither of the other event handlers in WaitForCancel can execute. The with statement block executes after the exit action of the exiting state. The event default is a special event that is internally generated by the P# runtime.

3.2. Substitution

A machine can be used to model another machine for systematic testing, using the *substitution* feature of the P# language. To achieve this, the programmer has to use the following special create statement:

```
create Foo () models Bar;
```

In this case, the Foo machine is used to model the Bar machine. During systematic testing, the Foo machine will be created, whereas during real execution, the Bar machine will be created.

Likewise, a method can model another method for systematic testing. The concept behind this is similar to how a machine can model another machine. The following special call statement can be used:

```
Foo() for Bar;
```

In this case, the Foo method will be called instead of the Bar method during systematic testing, wheras the Bar method will be called during real execution.

3.3. Failure detection protocol

The failure detection protocol is based on the interaction of two types of machines, FailureDetector and Node, which are shown below. The code of the FailureDetector machine presents advanced features of P#, including push transitions and pop statements, deferred events, and monitors.

```
// FailureDetector.psharp
namespace FailureDetector {
  // request from failure detector to node
  event Ping;
  // response from node to failure detector
  event Pong;
  // register a client for failure notification
  event RegisterClient;
  // unregister a client from failure notification
  event UnregisterClient;
  // local events for control transfer within failure detector
  event RoundDone;
  event TimerCancelled;

  machine FailureDetector {
    List<machine> nodes;                  // nodes to be monitored
    Dictionary<machine, bool> clients;    // registered clients
    int attempts;                         // number of Ping attempts made
    Dictionary<machine, bool> alive;      // set of alive nodes
    Dictionary<machine, bool> responses; // collected responses in one round
    machine timer;

    start state Init {
      entry {
        this.nodes = new List<machine>();
        this.clients = new Dictionary<machine, bool>();
        this.alive = new Dictionary<machine, bool>();
        this.responses = new Dictionary<machine, bool>();
        this.nodes = payload as List<machine>;
        this.InitializealiveSet();
        this.timer = create Timer ( this );
        raise Unit;
      }

      on RegisterClient do {
        this.clients[(machine)payload] = true;
      };

      on UnregisterClient do {
        if (this.clients.ContainsKey((machine)payload)) {
          this.clients.Remove((machine)payload);
        }
      };

      on Unit push SendPing;
    }

    state SendPing
    {
      entry {
        this.SendPings(); // send Ping events to machines that have not responded
        send this.timer, Start ( 100 ); // start timer for intra-round duration
      }

      on Pong do {
        // collect Pong responses from alive machines
        if (this.alive.ContainsKey((machine)payload))
        {
          this.responses[(machine)payload] = true;
          if (this.responses.Count == this.alive.Count) {
            // status of alive nodes has not changed
            send this.timer, Cancel;
            raise TimerCancelled;
          }
        }
      };

      on TimerCancelled push WaitForCancelResponse;

      on Timeout do {
        // one attempt is done
        this.attempts++;
        // maximum number of attempts per round == 2
        if (this.responses.Count < this.alive.Count && this.attempts < 2) {
          raise Unit; // try again by re-entering SendPing
        } else {
          this.CheckaliveSet();
          raise RoundDone;
        }
      };

      on Unit goto SendPing;
      on RoundDone goto Reset;
    }

    state WaitForCancelResponse {
      defer Timeout, Pong;

      on CancelSuccess do {
        raise RoundDone;
```

```
      };

      on CancelFailure do {
        pop;
      };
    }

    state Reset {
      entry {
        // prepare for the next round
        this.attempts = 0;
        this.responses.Clear();
        send this.timer, Start ( 1000 ); // start timer for inter-round duration
      }

      on Timeout goto SendPing;
      ignore Pong;
    }

    void InitializealiveSet() {
      foreach (var node in this.nodes) {
        this.alive.Add(node, true);
      }
    }

    void SendPings() {
      foreach (var node in this.nodes) {
        if (this.alive.ContainsKey(node) && !this.responses.ContainsKey(node)) {
          monitor Safety, MPing ( node );
          send node, Ping ( this );
        }
      }
    }

    void CheckaliveSet() {
      foreach (var node in this.nodes) {
        if (this.alive.ContainsKey(node) && !this.responses.ContainsKey(node)) {
          this.alive.Remove(node);
        }
      }
    }
  }
}
```

The state `Init` of the `FailureDetector` machine illustrates the use of `push` transitions. After appropriately initializing fields in the entry action, the raised event `Unit` is handled by *pushing* state `SendPing` on top of the current state `Init`. Thus, the use of `push` transitions creates a stack of states encoding the control of a state-machine. As long as state `Init` is on the stack, event handlers declared inside it are available to be executed regardless of changes in any states above it. The event handlers for `RegisterClient` and `UnregisterClient` are such available handlers. A `push` transition from Init to SendPing (rather than a goto transition) enables the declaration of these handlers in one place (the Init state) and their reuse everywhere else in FailureDetector. In addition to offering reuse of event handlers, push transitions also enable reuse of protocol logic for handling specific interactions with other machines. We later explain this use of a push transition in handling the interaction between FailureDetector and Timer.

When the state SendPing of the FailureDetector machine is entered, its entry action sends Ping events to all alive nodes that have not yet responded with a Pong event and starts a timer with a timeout value of 100ms. The machine stays in this state collecting Pong responses until either all alive nodes have responded or a Timeout event is dequeued. If each alive node has responded with a Pong before Timeout is dequeued, the timer is canceled and the event TimerCancelled is raised. Otherwise, the handler of Timeout is executed to determine whether another attempt to reach the potentially alive nodes should be made. If the number of attempts has reached the maximum number of attempts (2 in this example), then the nodes have failed.

The state SendPing of FailureDetector illustrates the use of a push transition to factor out the logic for handling timer cancelation. When TimerCancelled is raised after canceling the timer because all alive nodes have responded with Pong, the handler on TimerCancelled push WaitForCancelResponse pushes the state WaitForCancelResponse on top of the state SendPing. The state WaitForCancelResponse handles the interaction with the timer subsequent to its cancelation, returning control back to SendPing afterwards. The timer may respond with either CancelSuccess or CancelFailure. In the former case, the event RoundDone is raised which is not handled in state TimerCancelled causing the state to be popped and letting the state SendPing handle the RoundDone event. In the latter case, the pop statement executes causing TimerCancelled to be popped and a fresh event being dequeued in the state SendPing.

The state WaitForCancelResponse uses the code defer Timeout, Pong to indicate that it is not willing to handle Timeout and Pong states in this state. Therefore, while the machine is blocked in this state, the P# runtime does not dequeue these two events, instead *skips* them to retrieve any other event in the input queue.

When the FailureDetector machine is blocked in the state WaitForCancelResponse, its stack has three states on it. Starting from the bottom of the stack, these states are Init, SendPing, and WaitForCancelResponse. The Init state specifies do handlers for RegisterClient and UnregisterClient, the SendPing state specifies do handlers for Pong and Timeout, and the WaitForCancelResponse state defers Pong and Timeout. The dequeue logic for P# allows the execution of do handlers specified anywhere in the stack *unless* deferred in a state above. Therefore, in the state WaitForCancelResponse, the events RegisterClient and UnregisterClient may be dequeued, but the events Timeout and Pong may not.

```
// Node.psharp
namespace FailureDetector {
  machine Node
  {
    start state WaitPing
    {
      on Ping do
      {
        monitor Safety, MPong ( this );
        send (machine)payload, Pong ( this );
      };
    }
  }
}
```

```
// Safety.psharp
namespace FailureDetector {
  event MPing;
  event MPong;

  monitor Safety {
    Dictionary<machine, int> Pending;

    start state Init {
      entry {
        this.Pending = new Dictionary<machine, int>();
      }

      on MPing do {
        if (!this.Pending.ContainsKey((machine)payload)) {
          this.Pending[(machine)payload] = 0;
        }

        this.Pending[(machine)payload] = this.Pending[(machine)payload] + 1;
        assert (this.Pending[(machine)payload] <= 3);
      };

      on MPong do {
        assert (this.Pending.ContainsKey((machine)payload));
        assert (this.Pending[(machine)payload] > 0);
        this.Pending[(machine)payload] = this.Pending[(machine)payload] - 1;
      };
    }
  }
}
```

P# also allows programmers to write *assertions* that express invariants on the local state of a machine. This can be achieved using the `assert` statement. However, it is often useful to be able to write assertions about state across machines in a program. P# provides *monitors* that enable writting such specifications. Consider the problem of specifying that the difference in the number of `Ping` events sent to any machine can never be three more than the number of `Pong` events sent by it. This specification can be encoded using the monitor machine `Safety`. This monitor maintains the difference between the number of `Ping` and `Pong` events per machine in a C# dictionary and asserts that this number can be at most three. The failure detection protocol communicates with this monitor by sending it `MPing` and `MPong` events using a statement such as `monitor Safety, MPong ( this )`.

3.4. Test driver machine

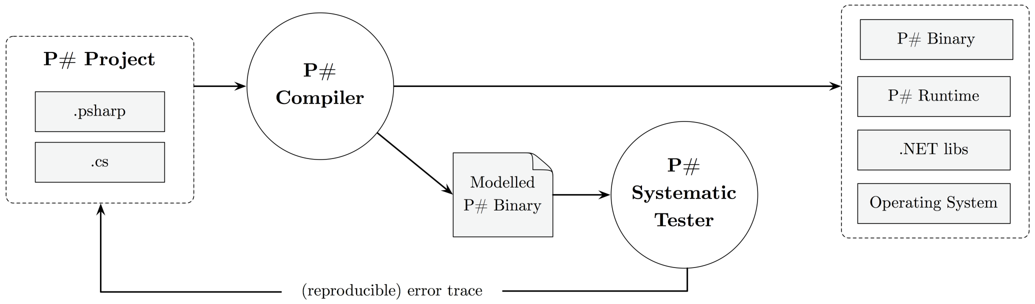
The machine `Driver` shows how to write a test driver to test the failure detection protocol. This machine models a client of the protocol. This client creates a few nodes to be monitored, creates an instance of the `Safety` monitor, an instance of `FailureDetector`, registers itself with the created instance of `FailureDetector`, and then enqueues the special `halt` event to each node created by it to terminate that node. Thus, the `Driver` machine creates a finite test program for the failure detection protocol.

```
// Driver.psharp
namespace FailureDetector {
  model Driver {
    machine FailureDetector;
    List<machine> NodeSeq;
    Dictionary<machine, bool> NodeMap;

    start state Init {
      entry {
        this.NodeSeq = new List<machine>();
        this.NodeMap = new Dictionary<machine, bool>();
        this.Initialize();
        create Safety ();
        this.FailureDetector = create FailureDetector ( this.NodeSeq );
        send this.FailureDetector, RegisterClient ( this );
        this.Fail();
      }
    }

    void Initialize() {
      for (int i = 0; i < 2; i++) {
        var node = create Node ();
        this.NodeSeq.Add(node);
        this.NodeMap.Add(node, true);
      }
    }

    void Fail() {
      for (int i = 0; i < 2; i++) {
        send this.NodeSeq[i], halt;
      }
    }
  }

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

Even though the test program encoded by machine `Driver` is finite, it can generate an enormous number of behaviors resulting primarily from the concurrent execution of a number of state machines: one `Driver` machine, two `Node` machines, one `FailureDetector` machine, and one `Timer` machine. At each step in an execution when a non-local action (creation of a machine or sending an event from one machine to another) is about to execute, there is a choice of picking *any* one of these machines to execute. The testing infrastructure embedded in the P# compiler *systematically enumerates* these behaviors; if an exception is raised or an assertion is violated, a path to the error is reported to the programmer.

The style of specifying test programs in the manner described above results in a compact description of a large set of test executions, considerably reducing the programmer's effort in specifying and generating them. For example, even though the `Driver` machine sends a `halt` event to each `Node` machine immediately after creating them, these send actions may be arbitrarily *delayed* if a nondeterministic scheduler chooses to execute the other machines in the program instead. The P# systematic tester has the capability to enumerate such behaviors to explore *tricky* interleavings between event handlers.

4. Tools



The above figure shows a typical P# workflow. The programmer initially creates a P# application and then uses the compiler to parse, compile and systematically test this application. This process repeats until all bugs are discovered and fixed.

4.1. Compilation

The compiler executable is called `PSharpCompiler.exe`. It accepts a P# solution, which can contain a number of `.psharp` and `.cs` files, and various optional arguments as input. The options can be discovered by running `PSharpCompiler.exe /?` from the command line.

The compiler is based on a three-phase compilation approach: it initially parses the P# surface syntax and reports any P#-related syntax errors; it then rewrites the original P# program into an intermediate C# representation; it finally uses the Roslyn compiler to parse and compile the intermediate C# program to an executable file (`.exe`) or library (`.dll`), and link it with the P# libraries and C# system libraries. Any C#-specific compile errors will be reported in this final Roslyn-based compilation phase.

The P# compiler can be invoked on a P# solution with the following command:

```
.\PSharpCompiler.exe /s:${SOLUTION_PATH}\${SOLUTION_NAME}.sln
```

To build only a single specific project from a given solution use the `/p:${PROJECT_NAME}` command line option.

4.2. Execution

To execute the compiled program, run the executable that is produced by `PSharpCompiler.exe` (found in the directory specified by the P# project). If the compiled program is a library, then link the library as in any typical .NET project. A typical P# application consists of one or more P# machines communicating asynchronously with each other by sending and receiving events. The P#

runtime is responsible for handling all the underlying asynchrony when a machine creates a new machine or sends an event to a target machine.

4.3. Systematic Testing

Once the programmer fixes all the statically discovered errors, the compiler can be used to systematically test the P# application. This is achieved by running the compiler in *bug-finding* mode. In this mode, the systematic tester takes control of the underlying schedule and systematically explores event handling interleavings to discover bugs in the P# program.

To switch to bug-finding mode the command-line option `/test` must be given when invoking the P# compiler.

By default, the systematic tester will explore a single iteration using an exploration depth bound of 1000. This depth bound is the maximum number of scheduling steps that the systematic tester will take while exploring a single schedule of the program. To increase the depth bound use the command line option `/db:x` where `x` is a positive integer. To increase the number of schedules to be explored (from program start to a terminal state) use the command line option `/i:x` where `x` is a positive integer.

If a dynamic error is discovered, the systematic testing will terminate, and a complete error trace starting from the initial state of the program is dumped on the disk in a text file (in the same directory as the compiled binaries of the P# application). The programmer can fix the found bug and run the compiler and systematic testing again on the modified program. Search statistics are also printed during and at the end of the systematic testing.

5. Glossary

Types	
Syntax	**Description**
`machine`	A reference to a P# machine.
type	The set of all possible C# types (either primitive, such as `int` and `bool`, or custom).

Machines and Events	
Syntax	**Description**
`default`	A pseudo-event that may be taken when no other event is triggered. Taking a `default` event sets `trigger` and `payload` to null.
`halt`	A predefined event that causes a machine to shutdown. The user implements a machine's response to `halt`. A machine is shutdown when its state stack becomes empty with `trigger` == `halt`.
`event` e [{`assert` | `assume`} k];	Declares an event.[a]
`machine` M [{`assert` | `assume`} k] `{ … }`	Declares a machine.[b]
`model` M [{`assert` | `assume`} k] `{ … }`	Declares a model machine.[b]
`monitor` M `{ … }`	Declares a property monitor.

[a] An event e may optionally be annotated with `assert k` or `assume k`. The former specifies that there must not be more than k instances of e in the input queue of any machine. The latter specifies that during systematic testing, an execution that increases the cardinality of e beyond k in some queue must not be generated. Both compile to an assertion in code generated for execution.

[b] A machine or model M may optionally be annotated with `assert k` or `assume k`. Their meaning is similar to that of the corresponding annotation for an event except that the bound k refers to the total number of events in the input queue of M.

States, State Variables, and Functions (declared inside machines)	
Syntax	**Description**
`this`	Refers to the current instance of a machine.
`trigger`	Refers to the event causing the current entry or exit of a state; it is initially `null`.
`payload`	Refers to the payload of the event causing entry or exit of a state; initially the value passed to the `create` operator.
[`start`] `state` S `{ … }`	Declares a (starting) state S.
[`model`] method declaration	Declares a (model) C# method with zero or more formal parameters.

State Actions and Transitions (declared inside states)	
Syntax	**Description**
`entry` A`;`	An action A to be executed whenever this state is entered.
`entry { … }`	A code block to be executed whenever this state is entered.
`exit` A`;`	An action A to be executed whenever this state exits.
`exit { … }`	A code block to be executed whenever this state exits.
`defer` e_1, …, e_n;	A list of events to skip while examining the queue for an event to handle.
`ignore` e_1, …, e_n;	A list of events to immediately remove while examining the queue for an event to handle.
`on` e_1, …, e_n `do` A;	A list of events to immediately remove while examining the queue for an event to handle. Each removal causes execution of A.
`on` e_1, …, e_n `do { … }`	A list of events to immediately remove while examining the queue for an event to handle. Each removal causes execution of code block.
`on` e_1, …, e_n `goto` S [`with` A];	A handler triggered whenever some e_i is encountered in the queue. Causes removal of e_i from queue, execution of exit function, (execution of A), and entry into state S.
`on` e_1, …, e_n `goto` S `with { … }`	A handler triggered whenever some e_i is encountered in the queue. Causes removal of e_i from queue, execution of exit function, execution of code block, and entry into state S.
`on` e_1, …, e_n `push` S;	A handler triggered whenever some e_i is encountered in the queue. Causes removal of e_i from queue, pushing of current state onto state stack, and entry into state S.

P# Statements	
Syntax	**Description**
`create` M (v_1, …, v_n);	Creates a new instance of a machine or monitor with type M, and with payload set to (v_1, …, v_n).
`raise` ev [(v_1, …, v_n)];	Causes immediate handling of event expression ev (and sets payload to (v_1, …, v_n)).
`send` dst, ev [(v_1, …, v_n)];	Sends to machine expression dst event expression ev (with payload to (v_1, …, v_n)).
`pop`;	Pops this state from the state stack and returns to the parent state. Fails if the state stack is empty and the trigger is not the `halt` event.
`assert` v;	Fails if the expression v does not evaluate to `true`.