BaZinga: Towards Symbolic Analysis of P programs

Surya Teja Chavali[1], Akash Lal[2]

[1] cs13b1028@iith.ac.in, *Indian Institute of Technology Hyderabad*, India
[2] akashl@microsoft.com, *Microsoft Research*, India

1 Introduction

P [3] is a language for programming asynchronous event-driven systems. P allows the programmer to specify the system as a collection of interacting state machines, which communicate with each other using events. P unifies modeling and programming into one activity for the programmer. Not only can a P program be compiled into executable code, but it can also be validated using systematic testing, via the Zing [1] analysis tool. P has been used to implement and validate the USB device driver stack that ships with Microsoft Windows 8 and Windows Phone. P is also suitable for the design and implementation of networked, embedded, and distributed systems.

The motivation behind our work is to extend the analysis capabilities offered to the P programmer. Zing is an explicit-state model checker that covers multiple possible behaviors of the program through explicit enumeration. It has been designed to explore concurrent systems very efficiently. However, it is limited in covering the space of data values. The P programmer, thus, has to use concrete values when designing a test harness. The only exception is the use of Boolean choice, but this is not sufficient in many cases. We would like the P programmer to work with symbolic values, e.g., to test their system when n messages of type T_1 and m messages of type T_2 are sent to it by the environment, where n and m are symbolic values.

To allow symbolic analysis of P programs, our attempt is to first compile it to a Boogie [2] program and then apply the Corral [4] analysis tool. Corral is able to handle concurrency as well as symbolic values. Our work will allow for a scientific comparison of two approaches to program verification: symbolic verification via Corral and state-space exploration via Zing. This report details the design of the translator from P to Boogie.

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

Programming model P provides primitives for creating machines, sending events from one machine to another, and writing assertions about system properties. Each machine has an input queue, a state stack, 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.

In the rest of this article, we first examine the mapping of basic features of a P program to the corresponding Boogie Program. We will then outline the process of the translation itself.

2 Mapping basic features of a P program

A P program is a collection of `event` and `machine` declarations.

2.1 Types

All P variables are translated to Boogie variables of a single (user-defined) type `PrtRef`, and the types themselves are mapped to Boogie constants of a type called `PrtType`. Compound types such as maps and sequences are hashed to unique Boogie constants of type `PrtType`, and represented as `PrtTypeMap1`, `PrtTypeMap2`, etc. for maps and `PrtTypeSeq1`, `PrtTypeSeq2`, etc. for sequences. The function `PrtDynamicType` returns the P type of a `PrtRef`.

```
type PrtType;
const unique PrtTypeNull: PrtType; //null
const unique PrtTypeInt: PrtType; //int
const unique PrtTypeBool: PrtType; //bool
const unique PrtTypeMachine: PrtType; //machine
const unique PrtTypeEvent: PrtType; //event
const unique PrtTypeTuple1: PrtType //Tuple of length 1
const unique PrtTypeTuple2: PrtType //Tuple of length 2
const unique PrtTypeTuple3: PrtType //Tuple of length 3
//Map of some type. The number 5 depends on the program at hand.
const unique PrtTypeMap5: PrtType
const unique PrtTypeMap6: PrtType //Map of a different type.
const unique PrtTypeSeq4: PrtType //Sequence of some type.

type PrtRef;
const unique null: PrtRef; //null
const unique PrtTrue: PrtRef; //true
const unique PrtFalse: PrtRef; //false
function PrtDynamicType(PrtRef) :PrtType;
function PrtIsNull(PrtRef) : bool;
```

The function `PrtConstructFromInt` constructs a `PrtRef` from an `int`, and `PrtFieldInt` retrieves an integer from a `PrtRef`. An axiom is used to ensure that these functions are inverses of each other. Likewise for non-composite types like `event`s, `machine`s and `bool`s. The translation of composite types is more complicated.

```
function PrtConstructFromInt(int) : PrtRef;
function PrtConstructFromEventId(int) : PrtRef;
function PrtConstructFromMachineId(int) : PrtRef;
function PrtConstructFromBool(bool) : PrtRef;

function PrtFieldInt(PrtRef) : int;
function PrtFieldBool(PrtRef) : bool;
function PrtFieldMachine(PrtRef) : int;
function PrtFieldEvent(PrtRef) : int;

procedure {:allocator} AllocatePrtRef() returns (x: PrtRef);
```

Tuples Tuples are translated in accordance with their length; an `(int)` will be a `PrtTypeTuple1`, and **both** `(int, bool)` **and** `(int, int)` are translated to `PrtTypeTuple2`. The function `PrtFieldTuple1` will return a `PrtRef` corresponding to the value at position 1 of the argument, and so on. Note that we start counting tuple positions with 0.

```
function PrtFieldTuple0(PrtRef) : PrtRef;
function PrtFieldTuple1(PrtRef) : PrtRef;
```

Table 1: Tuple Operations

Original	**Translation**
`x = t.2;`	`call x := PrtFieldTuple2(t);`

`t = (1, 2, E, true);`	`tmpRhsValue_0 := PrtConstructFromInt(1);`
	`tmpRhsValue_1 := PrtConstructFromInt(2);`
	`tmpRhsValue_0 := PrtConstructFromEventId(E);`
	`tmpRhsValue_0 := PrtTrue;`
	`call t := AllocatePrtRef();`
	`assume PrtDynamicType(t) == PrtTypeTuple4;`
	`assume PrtFieldTuple0(t) == tmpRhsValue_0;`
	`assume PrtFieldTuple1(t) == tmpRhsValue_1;`
	`assume PrtFieldTuple2(t) == tmpRhsValue_2;`
	`assume PrtFieldTuple3(t) == tmpRhsValue_3;`

Maps Map types are made to correspond to integers: a `map[int, int]` could be a `PrtTypeMap2`, for instance - these type names are entirely a function of the program being translated. A map is represented as two arrays: one for keys and another for values. The functions `PrtFieldMapKeys` and `PrtFieldMapValues` yield the corresponding array for a given `PrtRef`. A number of custom functions are used for map functionality: `ReadMap`, `WriteMap`, `InsertMap`, etc., and are detailed below.

```
function PrtFieldMapKeys(PrtRef) : [int]PrtRef;
function PrtFieldMapValues(PrtRef) : [int]PrtRef;
function PrtFieldMapSize(PrtRef) : int;
procedure MapContainsKey(map: PrtRef, key: PrtRef)returns (v: PrtRef)
procedure ReadMap(map: PrtRef, key: PrtRef) returns (value: PrtRef)
procedure MapGetKeys(map: PrtRef) returns (seq: PrtRef)
procedure MapGetValues(map: PrtRef) returns (seq: PrtRef)
procedure WriteMap(map: PrtRef, key: PrtRef, value: PrtRef) returns (nmap: PrtRef)
procedure InsertMap(map: PrtRef, key: PrtRef, value: PrtRef) returns (nmap: PrtRef)
procedure RemoveMap(map: PrtRef, key: PrtRef) returns (nmap: PrtRef)
```

Table 2: Map Operations

Original	Translation
`i = sizeof(m);`	`i := PrtFieldMapSize(t);`
`kl = keys(m);`	`kl := PrtFieldMapKeys(t);`
`vl = values(m);`	`i := PrtFieldMapValues(t);`

`m = default(map[int, int]);`	`call m := AllocatePrtRef();`
	`//Assuming 3 is the index of map[int, int]`
	`assume PrtDynamicType(m) == PrtTypeMap3;`
	`assume PrtFieldMapSize(m) == 0;`

`b = k in m;`	`call b := MapContainsKey(m, k);`
`m[k] = v;`	`call m := WriteMap(m, k, v);`
`m += (k, v);`	`call m := InsertMap(m, k, v);`
`m -= k;`	`call m := RemoveMap(m, k);`

Sequences Sequence types, like map types, are made to correspond to integers by hashing on the type. A sequence is represented as an array of values. The function `PrtFieldSeqStore` yields the corresponding array for a given `PrtRef`. A number of custom functions are used for sequence functionality: `ReadSeq`, `WriteSeq`, `InsertSeq`, etc., and are detailed below.

```
function PrtFieldSeqStore(PrtRef) : [int]PrtRef;
function PrtFieldSeqSize(PrtRef) : int;
procedure WriteSeq(seq: PrtRef, index: int, value: PrtRef) returns (nseq: PrtRef)
procedure RemoveSeq(seq: PrtRef, index: int) returns (nseq: PrtRef)
procedure InsertSeq(seq: PrtRef, index: int, value: PrtRef) returns (nseq: PrtRef)
```

Table 3: Sequence Operations

Original	**Translation**
`i = sizeof(s);`	`i := PrtFieldSeqSize(s);`

`s = default(seq[int]);`	`call s := AllocatePrtRef();`
	`//Assuming 4 is the index of seq[int]`
	`assume PrtDynamicType(s) == PrtTypeSeq4;`
	`assume PrtFieldSeqSize(s) == 0;`

`s[i] = v;`	`call s := WriteSeq(s, PrtFieldInt(i), v);`
`s += (i, v);`	`call s := InsertSeq(s, PrtFieldInt(i), v);`
`s -= i;`	`call s := RemoveSeq(s, PrtFieldInt(i));`

2.2 Statements

Some statements in P are mapped to procedure calls in Boogie. They are shown below.

Table 4: Simple Statement Translation

Original	Translation
`x = a/b;`	`x := PrtConstructFromInt(PrtFieldInt(a) div PrtFieldInt(b));`
`m = this;`	`m := PrtConstructFromMachineId(thisMid);`

`j = i as int;`	`//Assume int is type No.1.`
	`call AssertIsType1(i);`
	`j := i;`

`flag = (a == b);`	`call flag := PrtEquals(a, b);`

2.3 Events and Machines

In Boogie, P events are represented as integers. Machine globals are represented as thread-local global variables in Boogie; machine functions and static functions as procedures. Local variables defined inside functions/ anonymous functions are procedure locals. All actions of `do`s, state transitions, `case`s in `receive` statements, and entry/exit actions of states(AnonFunDecls), are desugared to Boogie procedures.

For example, the program

```
event E: int;

machine M {
  var x: int;
  start state Init {
    on E do (payload: int) {
      x = foo(payload);
    }
  }
  state S { ... }
  fun foo(i: int): int {
    return i + 1;
  }
}
```

results in the translation

```
var{:thread_local} M_x: PrtRef;
procedure M_foo(M_foo_i: PrtRef) returns (ret: PrtRef)
{
  ret := PrtConstructFromInt(PrtFieldInt(M_foo_i) + 1));
}

procedure MachineThread_M(entryArg: PrtRef)
{
  //Initialize...
  while(true)
  {
    if(CurrState == 0) //Init
    {
     M_Init:
      call event, payload := Dequeue(thisMid, true);
      if(event == 0) /*E*/ { call M_x := foo(payload); }
      else if (event == 1) /*Halt*/ { return; }
    }

    else if(CurrState == 1) { ... }

    else { assume false; }
  }
}
```

2.4 Translation of Monitors

Monitors are a special case in P, because of a multitude of reasons:

- Monitors may not use the `this` keyword, perform non-deterministic choice, create machines, have null or push transitions, defer events, or execute `send`, `receive`, `pop`, or `monitor` statements.
- Monitors must be called synchronously - the corresponding monitor must be called immediately after an event is sent.

For these reasons, monitor translation is treated separately in the following ways:

- Monitor procedures are represented as `Monitor_M`, and do not run on a different thread.
- Monitors do not possess a State Stack, or an event queue.
- Monitor globals are absolute globals, not thread local ones.
- An unhandled event is not an error.
- The `send()` procedure calls the corresponding monitor procedures before en-queuing the event to the queue of the target machine.
- Monitor type checking is more stringent to enforce the above rules.

An example is shown on the next page.

```
spec M monitors Ping, Pong {
var pingCount: int;
var pongCount: int;
  start state ExpectPing {
    on Ping goto ExpectPong;
  }
  state ExpectPong {
    on Pong goto ExpectPing;
  }
}
```

results in the translation

```
var M_pingCount: PrtRef;
var M_pongCount: PrtRef;

procedure Monitor_M(event: int, payload: PrtRef)
{
  if(M_CurrState == 0) // M_ExpectPing
  {
   M_ExpectPing:
    if(event == 0) // Ping
    {
      call M_ExpectPing_exit47(null);
      call payload := M_ExpectPing_on_Ping_goto_M_ExpectPong48(payload);
      M_CurrState := 1;
      call M_ExpectPong_entry51(payload);
    }
    else if(event == 1){} //Nothing to do for Pong.
    else
    {
      assert false;
    }
  }
  else if(M_CurrState == 1) // M_ExpectPong
  {
   M_ExpectPong:
    if(event == 1) // Pong
    {
      call M_ExpectPong_exit51(null);
      call payload := M_ExpectPong_on_Pong_goto_M_ExpectPing52(payload);
      M_CurrState := 0;
      call M_ExpectPing_entry47(payload);
    }
    else if(event == 0){} //Nothing to do for Ping.
    else
    {
      assert false;
    }
  }
}
```

2.5 How the dequeue works

The working of the dequeue is best illustrated by means of the following code:

```
procedure Dequeue(block: bool) returns (event: int, payload: PrtRef)
{
    if(eventRaised)
    {
        eventRaised := false;
        event := raisedEvent;
        payload := raisedEventPl;
        return;
    }
    int head := MachineInboxHead[thisMid];
    int tail := MachineInboxTail[thisMid];
    int ptr := head;
    int event := 0 - 1;
    PrtRef payload := null;
    if(!block && head > tail) { return; } //Handle non-blocking case "on null do"
    while(ptr <= tail)
    {
        event := MachineInboxStoreEvent[thisMid][ptr];
        if(event >= 0 && ignoreEvents[CurrState][event])
        {
            // ignore the event
            MachineInboxStoreEvent[thisMid][ptr] := 0 - 1;
            MachineInboxStorePayload[thisMid][ptr] := null;
        }
        else if(event >= 0 && !deferEvents[CurrState][event])
        {
            // dequeue
            q := machineEvToQCount[thisMid][event];
            machineEvToQCount[thisMid][event] := q - 1;
            MachineInboxStoreEvent[thisMid][ptr] := 0 - 1;
            payload := MachineInboxStorePayload[thisMid][ptr];
            // book-keeping for head and tail.
            break;
        }
        ptr := ptr + 1;
        event := 0 - 1;
    }
    // block
    assume (event >= 0);
}
```

The translation for a `receive` statement does **not** call the above procedure; it probes the event queue itself in a similar fashion. The reasons for this are manifold:

- The semantics of `receive` statement defers all events that are not explicitly handled - unlike a state event handler, which throws an error.
- The registered, ignored and deferred event maps of the machine must not be corrupted, but a `receive` statement requires them to be reset.

3 How the translation proceeds

3.1 Pre-processing: P to P transformations

This phase, largely written in C#, is done via a custom `SymbolTable` class, which essentially maintains a stack of nested scopes. It proceeds in the following stages.

1. **Name Mangling** Because the translation converts machine globals to thread local globals, names of all states, functions, and variables are mangled into the following format to avoid naming conflicts:

   <machine_name>_≤function_name>_<variable_name>

2. **Converting Anonymous Functions to Functions** This is a complicated step, as anonymous functions must capture the local scope as well as the global scope. Translation proceeds by capturing local scope in a tuple, `env`, passed as an argument to the function created, and restoring the variables after the function call:

```
machine M {
  var mach_global: int;
  fun foo() {
    var fun_local: int;
    receive {
      case E:(payload: int) {
        fun_local += mach_global;
      }
      ... }}}
```

   results in

```
machine M {
  var mach_global: int;
  fun M_foo_recv_case_E(payload: int, Main_foo_env:(int, int)) {
    var M_foo_recv_case_E_fun_local: int;
    var M_foo_recv_case_E_mach_global: int;

    M_foo_recv_case_E_fun_local = Main_foo_env.0;
    M_foo_recv_case_E_mach_global = Main_foo_env.1;

    M_foo_recv_case_E_fun_local += (M_foo_recv_case_E_mach_global +
        payload);
    return (M_foo_recv_case_E_fun_local,
        M_foo_recv_case_E_mach_global);
  }

  fun foo() {
    var fun_local: int;
    var Main_foo_env: int;
    receive {
      case E:(payload: int) {
        Main_foo_env = (fun_local, mach_global);
        Main_foo_env = M_foo_recv_case_E(payload, Main_foo_env);
        fun_local = Main_foo_env.0;
        mach_global = Main_foo_env.1;
      }
    ... }}}
```

3. **Removing Reference parameters** proceeds in a similar fashion, and is implemented in F#. An example is shown below.

```
fun F(a ref: int, b : int) : int
{
  a = a + b;
  return b + 1;
}
fun caller()
{
  ...
  x = F(a ref, b);
  ...
}
```

becomes

```
fun F(a : int, b : int) : (int, int)
{
  a = a + b;
  return (b + 1, a);
}
fun caller()
{
  ...
  tmp = F(a, b);
  x = tmp.0;
  a = tmp.1;
  ...
}
```

4. **Removing `NamedTuples`** `NamedTuples` are desugared to `Tuples`, again using F#. For example,

```
...
var x: (i: int, b:bool, m: map[int, int]);
x.i = 2;
x.m += (5832, 4913);
...
```

becomes

```
...
var x: (int, bool, map[int, int]);
x.0 = 2;
x.2 += (5832, 4913);
...
```

3.2 Type-checking

We perform another type check of the P code, before we proceed to further break it down into a form assimilable directly into Boogie. The reason for this type-check is the fact that we use a slightly stronger typing system than P. Structural sub-typing is disallowed, e.g., we do not let a seq[int] value to be assigned to a variable of type seq[any]. This is done for efficiency of the resulting analysis, otherwise type casting would be costly. Further, we believe this imposes a minimal burden on the P programmer.

We do a runtime type checking of casts with `AssertIsType<int>` and of payloads of event variables via the method `AssertPayloadDynamicType()`.

3.3 Removing Side Effects

P statements are converted into a format amenable for directly mapping to Boogie. These statements are in the convenient form akin to a three-address code. The idea is that the resulting P statements should map directly to a single Boogie statement.

Some examples of this are shown in the table below.

Table 5: Removing Side Effects

Original	Side Effects Removed

`x = a + b.1 + c();`	`t1 = b.1;`
	`t2 = c();`
	`t3 = a + t1`
	`t4 = t2 + t3;`
	`x = t4;`

`y = f(g(x), 2);`	`t1 = g(x);` `y = f(t1, 2);`
`x = (y as int) + 1;`	`t1 = y as int;` `x = t1 + 1;`

`while(x1 != x2)`
`{`
`  ...`
`}`
	`b = !(x1 == x2);`
	`while(b)`
	`{`
	`  ...`
	`  b = !(x1 == x2);`
	`}`

`if($) {...}`	`t1 = $;` `if(t1) {...}`
`l.i = e;`	`l = (l.0, ..., l.(i-1), e, l.(i+1), ...);`

`l.0[e1] = e2;`	`t = l.0;`
	`t[e1] = e2;`
	`l.0 = t;`

Note: As a debugging feature of the translator, after each of the phases described above, the user can print the resulting P code, and it will be guaranteed to be a valid P program.

3.4 Output Boogie

This is the final phase of the translation, and the mapping of various P statements to the corresponding Boogie is straight-forward. Some examples are shown in the table below.

Table 6: Final translation

Original	Boogie Translation
`x = 1;`	`x := PrtConstructFromInt(1);`
`x += (0, 1);`	`call x := InsertSeq(x, 0, PrtConstructFromInt(1));`
`a = new M(x);`	`call a := newMachine_M(x);`
`send m, e, p;`	`call tmpEventID := AssertPayloadDynamicType(e, p);` `call send(m, tmpEventID, p);`

`if($) {...}`
`else {...}`
	`havoc tmpBool;`
	`b = PrtConstructFromBool(tmpBool);`
	`if(PrtFieldBool(b)) {...}`
	`else {...}`

`pop;`	`popped := true;`
	`call StateStackPop();`
	`return;`

`raise E, x;`	`eventRaised := true;`
	`raisedEvent := 1; //The integer corresponding to E`
	`raisedEventPl := x;`
	`return;`

`x = foo(a, b);`	`call x := foo(a, b);` `if(popped || eventRaised) { return; }`

Note: Because `pop` and `raise` statements trigger a behavior similar to raising an exception, (thread-local) global flags are set whenever a `pop`/`raise` occurs. After every function call, these flags are checked, so as to return to the machine thread context.

4 Testing the translation

The translation is tested against the regression tests of P. These include a rich battery of tests on the P framework, providing a large number of edge cases in the P semantics - with close to 140 tests that cover a variety of issues. Golden outputs as well as corral command-line flags for each of these tests have been generated, and stored in the directory Tst/RegressionTests of the repository. The translator comes with flags to run all regression tests, and compare against the golden outputs.
We assume **co-operative scheduling** in all of the Boogie programs that we generate, instead the pre-emptive scheduling of P, and the flag /cooperative must be passed to corral when used to verify the generated Boogie.
Further, we target only **safety bugs** for now; there is no attempt towards testing for liveness bugs.

References

1. T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: Exploiting program structure for model checking concurrent software. In *CONCUR 2004: 15th International Conference on Concurrency Theory*, volume 3170 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 2004.

2. R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

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.

4. Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri. A solver for reachability modulo theories. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 427–443, 2012.

A Auxiliary Functions and Procedures in the Translation

Some of these procedures can be found in the repository at BoogieExprTranslator/CommonBpl.bpl. Others are generated by the translator.

A.1 Types

`void AssertIsType<T>(x: PrtRef)` Procedure that asserts that x is of type T.

`PrtType PrtDynamicType(x: PrtRef)` Function that returns the type of x

`bool PrtIsNull(x: PrtRef)` Function that checks if x is null.

`PrtRef PrtConstructFrom<T>(x: PrtRef)` Function that constructs a PrtRef from x, of type T.

`int/bool PrtField<T>(x: PrtRef)` Function that returns the value of x if x is an `int`, `bool`, `machine`(machine ID), or `event`(integer corresponding to event)

`bool PrtEquals(a: PrtRef, b:PrtRef)` Procedure that returns true of a and b are equal. Does NOT compare maps/sequences.

A.2 Tuples

`PrtEqualsTuple<L>(a: PrtRef, b:PrtRef)` Procedure that compares two tuples field by field.

`PrtFieldTuple<i>(x: PrtRef)` Function that returns ith element of tuple x.

A.3 Maps

`int PrtFieldMapSize(m: PrtRef)` Function that returns number of elements in m.

`int[PrtRef] PrtFieldMapKeys(m: PrtRef)` Function that returns the array of keys of m.

`int[PrtRef] PrtFieldMapValues(m: PrtRef)` Function that returns the array of values of m.

`bool MapContainsKey(m: PrtRef, k: PrtRef)` Procedure that checks if k is present in m.

`PrtRef ReadMap(m: PrtRef, k: PrtRef)` Procedure that returns the value corresponding to k in m.

`PrtRef MapGetKeys(m: PrtRef)` Procedure that returns a sequence containing the keys of m.

`PrtRef MapGetValues(m: PrtRef)` Procedure that returns a sequence containing the values of m.

`PrtRef WriteMap(m: PrtRef, k: PrtRef, v: PrtRef)` Procedure that does m[k] = v;

`PrtRef InsertMap(m: PrtRef, k: PrtRef, v: PrtRef)` Procedure that does `m += (k,v);`
Throws an error if k is already present in m.

`PrtRef RemoveMap(m: PrtRef, k: PrtRef)` Procedure that does `m -= k;`
Throws an error if k is not present in m.

A.4 Sequences

`int PrtFieldSeqSize(s: PrtRef)` Function that returns the size of s.

`int[PrtRef] PrtFieldSeqStore(s: PrtRef)` Function that returns the array of elements of s

`bool SeqIndexInBounds(s: PrtRef, i: int)` Procedure that checks if 0 ¡= i ¡ PrtFieldSeqSize(s);

`PrtRef ReadSeq(s: PrtRef, i: int)` Procedure that returns `s[i]`

`PrtRef WriteSeq(s: PrtRef, i: int, v: PrtRef)` Procedure that does `s[i] = v;`
Throws an error if i is off bounds.

`PrtRef InsertSeq(m: PrtRef, i: int, v: PrtRef)` Procedure that does `s += (i,v);`
Throws an error if i is off bounds.

`PrtRef RemoveSeq(m: PrtRef, k: PrtRef)` Procedure that does `s -= i;`
Throws an error if i is off bounds.

A.5 Machines

`int newMachine_<M>(entryArg: PrtRef)` Procedure that creates new machine of type M, and mutates global book-keeping variables accordingly. Returns the machine ID of the newly created machine.

`void <M>_ProbeStateStack()` Procedure that probes the state state stack of M, performing implicit pop till such a time as a state that handles the event is found. Throws an error if there is no such state.

`<M>_CallEntryAction(arg: PrtRef, state: int)` Procedure that calls the entry action of state of M with the corresponding entryArg.

<M>_CallExitAction() Procedure that calls the exit action of CurrState of M.

MachineThread_<M>(entryArg: PrtRef) Procedure that instantiates the variables of a fresh instance of M, and performs the state transitions, send/receive, etc.

Monitor_<M>(entryArg: PrtRef) Procedure corresponding to the monitor M.

(int, PrtRef) Dequeue(mid: int, block: bool) Procedure that performs a blocking/non-blocking deque of the machine's inbox. Returns -1, null if the queue is empty on a non-blocking dequeue.

void InitializeInbox(mid: int) Procedure - sets mappings for `MachineInboxStoreEvent` and `MachineInboxStorePayload`.

void StateStackPush(state: int) Procedure for StateStackPush.

void StateStackPop() Procedure for pop.

void AssertMachineQueueSize() Procedure for asserting whether the machine's queue constraints are being followed.

void AssertEventCard(e: int) Procedure for asserting whether the event's queue constraints are being followed.

void Enqueue(e: int, pl: PrtRef) Procedure which checks for queue constraints, and performs enqueue if they're followed. Else throws an error.

monitor(e: int, pl: PrtRef) Procedure which calls the monitors for event e.

void send(mid:int, e: int, pl: PrtRef) Procedure which calls `monitor()`, and `Enqueue()`

PrtRef AssertPayloadDynamicType(e: int, pl: PrtRef) Procedure that asserts that pl is of the type e requires.