Formal Semantics of P in Maude

Manasvi Saxena (msaxena2@illinois.edu)
Nishant Rodrigues (nishant2@illinois.edu)

1 Problem Statement

P is a language for developing asynchronous systems. Components of a distributed system in P are modeled as state machines that interact via message passing. Systems written in P can use the P model checker to establish correctness. Furthermore, P generates performant C# code that can be used in production environments. P has been used successfully used components of complex large scale systems at AWS and Microsoft.

Even though P finds increasing adoption, the executable semantics of P are loosely defined in [1]. In this paper, we formally define the executable semantics of a fragment of the P programming language in Maude. Our fragment is complete enough to run the examples from the P tutorial.

2 Premliminaries

2.1 The P Programming Language

In this section, we briefly describe the P language. P is a Domain Specific Language for building asynchronous event driven programs [1]. A P program is a collection of machines that communicate via passing messages. Each machine has a collection of states, variables and actions. On receving an event, a machine may transition to another state by performing the action associated with said transition. P programs can be analyzed using a model checker, and compiled into executable code that can be used in a production environment. Favourable performance has allowed P to be used in large scale production environments at Microsoft and Amazon Web Services (AWS) [1].

We futher explain the features of P using an example program. Consider the following `server` machine written in P.

```
machine ServerMachine
sends eResponse ;
{
  var successful : bool ;
```

```
  start state Init {
    on eRequest do (payload : requestType) {
      successful = true ;
      send  payload . source
          , eResponse
          , (id = payload . id , success = successful) ;
    }
  }
}  .
```

The machine has a single state `Init`. On receiving an `eRequest` event, the machine sets updates its internal variable and sends the souce of the event an `eResponse` message.

Similarly, the `client` machine is defined as:

```
 machine ClientMachine sends eRequest ;
{ var a : int ;
  var server : ServerMachine ;
  var nextReqId : int ;
  var lastRecvSuccessfulReqId : int ;
  var index : int ;

  start state Init {
    entry
   {
      nextReqId = 1 ;
      server = new ServerMachine( .Exps ) ;
      goto StartPumpingRequests ;
    }
}

state StartPumpingRequests {
  entry {
    index = 0 ;
    while(index < 2)
    {
        send  server
            , eRequest
            , (source = this , id = nextReqId) ;
        nextReqId = nextReqId + 1 ;
        index = index + 1 ;
    }
  }

  on eResponse do (payload : responseType) {
       lastRecvSuccessfulReqId = payload . id ;
```

```
    }
  }
}
```

It sends event `eRequest` to the server machine, and after receiving a response, it updates its internal state to store the id of the last successful request.

3 P Semantics in Maude

3.1 Syntax

In this section, we briefly describe the syntax of various P constructs. For brevity, we only show some of the operators that make up the entire syntax.

3.1.1 Expressions

3.1.1.1 Arithmetic and Boolean Expression

```
sort Exp .
subsort Int < Exp .
subsort Id < Exp .

op _+_ : Exp Exp -> Exp [prec 33 gather (E e) format (d b o d)] .
op _/_ : Exp Exp -> Exp [prec 31 gather (E e) format (d b o d)] .

subsort Bool < Exp .
op _<=_ : Exp Exp -> Exp  [prec 37 format (d b o d)] .

--- Assignment Exp
op _=_ : Id Exp -> Exp [prec 39 format (d b o o)] .

--- Attributes Expression
op _._ : Exp Exp -> Exp [prec 38] .
```

Expressions in are semantics are parsed using the `Exp` sort. For instance, `_+_` is the `binary plus` operator in the language. `Assignment` takes an identifier on the LHS and an expression on the RHS.

3.2 Statements

3.2.1 Declarations

```
--- Expressions To Statements
op _; : Exp -> Stmt [prec 40] .

--- Stmt Composition
op __ : Stmt Stmt -> Stmt
  [prec 60 gather (e E) assoc id: .Stmt format (d ni d)] .
```

```
--- Variable Statements
op var_; : Exp -> Stmt [prec 40 format (y o o o)] .
```

3.2.2 Blocks

```
op {} : -> Block [format (b b o)] .
op {_} : Stmt -> Block [prec 39 format (d n++i n--i d)] .
```

3.2.3 If, While

```
op if(_)_else_ : Exp Stmt Stmt -> Stmt
  [prec 59 format (b so d d s nib o d)] .

op while(_)_ : Exp Block -> Stmt
  [prec 59 format (b so d d o d)] .
```

3.2.4 Machine Declaration

The following constructs are used to define machine in P. This allows us to parse the entire P program, which consists of a collection of machines as a `Stmt`.

```
sort MachineStmt .
subsort MachineStmt < Stmt .

op machine_;_ : Id Block -> MachineStmt [prec 41] .
op machine_sends_;_ : Id Exps Block -> MachineStmt [prec 41] .
```

4 Semantics

In this section, we describe the semantics of constructs introduced in section Section 3.1. Our semantics can be divided into two distinct phases: `static` and `dynamic`. During the static phase, we initialize the configuration, traverse the program, and populate appropriate parts of the configuration with information needed to execute the program. In the dynamic phases, the actual execution of the program occurs using information gathered in the static phases.

4.1 Configuration

We organize our configuration as an Associate Commutative (AC) soup of attributes. Attributes hold information about program execution. We briefly describe some imporant attributes:

- *pgm*: Holds the executing program.
- *machines*: Holds a set of *machine schemas* used to create instances of *machines* during execution.

```
subsort Attribute < Configuration .
op __ : Configuration Configuration -> Configuration
  [prec 65 assoc comm id: .Attribute] .

op (( pgm: _ )) : Stmt -> Attribute
  [prec 64 format(n d ++i -- d)] .
```

4.2 Static Phase

We now discuss the static phase.

4.2.1 Initialization

The first step in execution of any P program using our semantics. The initialization step which sets up the configuration by adding two attributes: - the `pgm` attribute holds the program to execute - the `machines` attribute holds a set of `Machine Schemas`, which we define in greater detail in section Section 4.3.

```
eq init(Ss)
 = ( pgm: Ss )
   ( machines:  .MachineSchemas )
```

4.3 Machine Schemas

P programs allow executing code to dynamically create instances of machines defined in the program via the `new` construct. This mechanism is analogous to classes and objects in Object Oriented Programming. A machine schema or definition in P can be thought of as a class, and instances of the schema its objects.

We organize machine schemas as an an attribute as follows:

```
op (( machines: _ )) : MachineSchemas -> Attribute
  [prec 64 format (n d ++ni d d)] .
```

In order to load a machine definition into the MachineSchemas cell, we define an operation called `statementToMachineSchema` as follows:

```
eq statementToMachineSchema(machine Id ; B)
 = statementToMachineSchema( B
                           , (machine: Id
                             , ( start: noid)
                             , ( states: .States)
                             , ( initState: .VariableMap)
                             ) ) .
eq statementToMachineSchema({ Ss }, Schema*)
 = statementToMachineSchema(Ss, Schema*) .
eq statementToMachineSchema( var X : T ; Ss
                           , (machine: Id
```

```
                                , (initState: Rho)
                                , MAttrs* ))
 = statementToMachineSchema(Ss
                              , (machine: Id
                                , (initState: (X |-> 0 Rho))
                                , MAttrs*)) .

eq statementToMachineSchema( (start state SId B) Ss
                              , (machine: MId
                                 , (start: noid)
                                 , MAttrs))
 = statementToMachineSchema( (state SId B) Ss
                              , (machine: MId
                                , (start: SId)
                                ,  MAttrs)) .
eq statementToMachineSchema( (state SId B) Ss
                             , (machine: MId
                                , (states:States)
                                , MAttrs* ))
= statementToMachineSchema( Ss
                            , ( machine: MId
                              , (states:
                                   ( statementToState( B
                                                     , ( state: SId
                                                       , (actions: .Actions)
                                                       , .SAttrs)))
                                 States)
                              , MAttrs*)) .
```

The `statementToMachineSchema` operator recursively traverses a machine definition and creates an appropriate sub-configuration for each machine. This subconfiguration comprises information regarding the machine such as its name, the set of state and the initial state.

4.4 Dynamic Phase

During the dynamic phase, we use the configuration that we setup during the static phase to execute the P program. We now describe some of the main execution rules in our semantics.

4.4.1 Expressions

We now describe how expression constructs are evaluated. To evaluate expressions, we define a construct called `eval` as follows:

```
 op eval : Exp VariableMap -> [Exp] .
-------------------------------------
```

```
eq eval(I, Rho) = I .
eq eval(Bool, Rho) = Bool .
eq eval(Id, (Id |-> Exp) Rho) = Exp .
eq eval(E1 < E2, Rho) = eval(E1, Rho) <Int eval(E2, Rho) .
eq eval(E1 + E2, Rho) = eval(E1, Rho) +Int eval(E2, Rho) .
```

The `eval` construct takes as input an Expression, and a mapping of identifiers to values, which acts as the environment under which the expression is evaluate. It produces the result of evaluating the expression under the specified environment. For example, `eq eval(Id, (Id |-> Exp) Rho) = Exp .` is the equation response for looking up the binding of an identifier in the state.

4.4.2 Statements

We now describe semantics for various statment constructs.

Assignment

```
eq ( instance: M
   , (code: X = E1 ; Ss)
   , (variables: X |-> E2 Rho)
   , IAttrs )
 = ( instance: M
   , (code: Ss)
   , (variables: X |-> eval(E1, X |-> E2 Rho) Rho)
   , IAttrs) .
```

On encountering an assingment statement of the form `X = E1` in the code attribute of an instance `M` of some machine, we first use the `eval` construct described in section Section 4.4.1 to reduce the rhs of the assignment construct. We then update the binding of the variable `X` in the `variables` attribute, which contains a mapping of identifiers to values, to the result of evaluating `E1`.

If

```
eq ( instance: M
    , (code: if (true) B1 else B2 Ss)
    , IAttrs )
 = ( instance: M
    , (code:  B1 Ss)
    , IAttrs ) .
eq ( instance: M
   , (code: if (false) B1 else B2 Ss)
   , IAttrs )
 = ( instance: M
   , (code:  B2                    Ss)
   , IAttrs ) .
ceq ( instance: M
    , (code: if (Exp) B1 else B2 Ss)
```

```
    , (variables: Rho)
    , IAttrs )
  = ( instance: M
    , (code:  if (eval(Exp, Rho)) B1 else B2 Ss)
    , (variables: Rho)
    , IAttrs )
  if notBool(Exp :: Bool) .
```

First, a conditional equation evaluates the condition until we get a boolean result. Then, based on the result of the evaluation of the condition, one of the equations for the appropriate true/false branch is applied.

New

The `new` statement corresponds to dynamic creation of an instance of a machine using the its machine schema.

```
rl ( counter: N )
   (instances:
    (instance: M
    , (code: X = new MId(Exps); Ss)
    , IAttrs
    )
    Machines
   )
=> ( counter: N +Int 1 )
   (instances:
    (instance: M
    , (code: X = N ;Ss)
    , IAttrs)
    Machines
    (new: MId, N, Exps)
   )
```

A global `counter` attribute is responsible used to assign ids to instances. Upon encountering the `new` construct, we increment the counter by one, and use the following construct to create an instance using as following:

```
op ((new: _, _, _)) : Id Nat Exps -> MachineInstance .
eq ( instances: (new: MId, N, Exps) Machines )
   ( machines: ( machine: MId
                , (start: StartId)
                , (initState: Rho)
                , MAttrs) ;; Schemas )
= ( machines: ( machine: MId
               , (start: StartId)
               , (initState: Rho)
               , MAttrs) ;; Schemas )
  ( instances: ( instance: N
```

```
                , (mid: MId)
                , enterState(StartId)
                , (state: noid)
                , (variables: (this |-> N) Rho)
                , (buffer: .Messages) )
    Machines ) .
```

The attribute labeled `new` takes as input the name of the machine to create, the id of the instance itself, and the expressions to pass to constructor. Note that on initializing the environment for the newly created instance, we bind the `this` variable to the value passed as an argument to the `new` attribute.

4.4.3 Inter-Instance Communication

We now discuss operations that facilitate inter instance communication. This occurs in P via message passing.

Send

```
ceq ( instance: M
    , (code: send Exp, Id, Exps ; Ss)
    , (variables: Rho)
    , IAttrs1 )
  = ( instance: M
    , (code: send eval(Exp, Rho), Id, evalExps(Exps, Rho) ; Ss)
    , (variables: Rho)
    , IAttrs1 )
 if notBool(Exp :: Nat) .

 rl ( instance: M
    , (code: send N,Exps ; Ss)
    , IAttrs1 )
    ( instance: N
    , (buffer: Messages)
    , IAttrs2 )
 => ( instance: M
    , (code: Ss)
    , IAttrs1 )
    ( instance: N
    , (buffer: (Messages, {Exps}))
    , IAttrs2 ) .
```

The `send` construct is used to send to another machine. The `send` construct requires the following: - The `identifier` to the target machine - The `event identifer` to send to the machine - A list of expressions dictating the data to be sent in the message.

For instance, consider the following send statement from the `ServerMachine` in

Section 2.1

```
send payload . source, eResponse, (id = payload . id , success = successful) ;
```

The statement above sends to the source of the payload, (the client machine), the event eResponse, where fields `id` and `success` of the message are bound to `payload.id` and `successful` respectively.

The rule for send uses the `eval` construct described in Section 4.4.1 to evaluate the expressions fully, and then adds the evaluated expression to the buffer fo the target machine.

4.4.4 Event Handlers

Events placed in the input buffer of an instance of a machine by the `send` construct described in section Section 4.4.3 are handled via event handlers.

We now describe the rule for handling events.

```
rl (instances: ( instance: M , (code: .Stmt)
                             , (buffer: { AId , Args }, Messages)
                             , (mid: MId)
                             , (state: SId)
                             , (variables: Rho)
                             , IAttrs1
                ) Machines )
   (machines: ( machine: MId
              , (states: ( state: SId
                         , (actions: (action: AId, Exps, Ss) Actions)
                         , SAttrs) States )
              , MAttrs) ;; Schemas )
=> (instances: ( instance: M , (code: Ss)
                             , (buffer: Messages)
                             , (mid: MId)
                             , (state: SId)
                             , (variables: bindArgs(Exps | Args | Rho ) )
                             , IAttrs1
                ) Machines )
   (machines: ( machine: MId
              , ( states: ( state: SId
                          , (actions: (action: AId, Exps, Ss) Actions)
                          , SAttrs ) States )
              , MAttrs ) ;; Schemas)
```

The rule performs the following steps:

- Dequeues an `event id, arguments` pair from the top of the `buffer` attribute.

- Looks up the appropriate action handling code in the machine schema and places it in the `code` attribute
- Updates the execution environment by binding the variables to values passed in the event via the `send` construct.

We now describe the `bindArgs` construct, which is responsible for updating the environment with appropirate values passed via `send`.

```
op bindArgs(_ | _ | _ ) : Exp Exps VariableMap -> VariableMap .
eq bindArgs( .Exps | Exps | Rho ) = Rho .
eq bindArgs(Id : T | Exps1 | (Id |-> Exps2) Rho )
 = (Id |-> Exps1) Rho .
eq bindArgs(Id : T | Exps1 | Rho )
 = (Id |-> Exps1) Rho [owise] .
```

The rules for `bindArgs` recursive traverse the list of values passed via the `send` construct, and updates the bindings of variables to said values.

5 Future Work

We identify several avenues for future work:

- Currently, we can only execute a subset of the P language. Note that this subset however, is enough to run programs in the P tutorial. The logical next step is to complete semantics for all remaining P constructs.
- Evaluating the performance of the P model checker against the maude search command.

6 Conclusion

In this paper, we presented semantics for the P programming language - a Domain Specific Language for asynchronous program that uses state machine to represent distributed system components. We formalized the semantics of a subset of the P language in Maude, and presented both the syntax (section Section 3.1) and the execution semantics (section Section 4). Finally, we presented several avenues for future research.

[1] A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajamani, and D. Zufferey, "P: Safe asynchronous event-driven programming," *SIGPLAN Not.*, vol. 48, no. 6, pp. 321–332, Jun. 2013, doi: 10.1145/2499370.2462184.