RB-003

An algebraic specification of message passing programming languages

Masaki Nakamura[1] Alaa Ismail El-Nashar[2] Kokichi Futatsugi[3]

1. Introduction

In this paper, we deal with parallel programming with message passing interface, where each process communicates with each other via functions which send and receive data between the processes. We describe a rewriting logic specification of a simplified parallel programming language supporting message passing functions in the algebraic specification language Maude [1] [4]. The parallel programming language we specify can be considered as a subset of Message Passing Interface (MPI) [11] [5], which is a message passing library interface specification. We show that both indeterminacy and deadlocks which may arise in parallel programs can be detected by using Maude system.

2. Maude in nutshell

A Maude specification consists of modules. Functional modules are used for describing abstract data types based on equations, and system modules are used for describing systems based on rewriting logic.

The following is an example of Maude functional modules:

```
fmod VAR is
  sort Var .
  ops a b c d e f g h i j k l m n o p q r
      s t u v w x y z pid np : -> Var .
endfm
```

Functional modules begin with `fmod` and end with `endfm`. The name of the above module is `VAR`. Sorts are declared after `sort`. The module `VAR` has the sort `Var`. By `ops` (or `op`), we can declare operation symbols. In `VAR`, operation symbols `a`, `b`, ... `pid`, `np` are declared. The rank of the operation symbol is, in general, given like $S_1\, S_2 \cdots S_{n-1}$ `->` S_n, where each S_i is a sort. The operation symbol takes terms whose sorts are $S_1\, S_2 \cdots S_{n-1}$ and forms a term of the sort S_n. The rank of the operation symbols in `VAR` are all `-> Var`, which means that those do not take any argument and form terms of `Var` by themselves. Such empty-argument operation symbols (or terms) are called constants.

2.1 Buitl-in modules

Maude supports built-in modules of fundamental data types, like Boolean, integers, strings, and so on. The built-in module `BOOL` is a special built-in module, which is imported by all user-defined modules implicitly. The built-in module `BOOL` has the sort `Bool`, the constants `true` and `false` of the sort `Bool`, and special polymorphic operation symbols: the equality predicates `_==_` and `_=/=_`, and the operation symbol `if_then_else_fi`. Underlines indicate the position of arguments in term expression. The equality predicates `_==_` and `_=/=_` are used for checking terms t_1 and t_2 are equal or not. The term t_1 `==` t_2 is reduced into `true` if they are equal, otherwise `false`. t_1 `=/=` t_2 is the negation of t_1 `==` t_2. The term `if` c `then` t_1 `else` t_2 `fi` is reduced into t_1 if c is true, otherwise t_2. Except above special operation symbols, `BOOL` has fundamental Boolean operation symbols `_and_`, `not_`, and so on. The following is the built-in module `BOOL`:

```
fmod BOOL is
  protecting TRUTH .
  op _and_ : Bool Bool -> Bool
             [assoc comm prec 55] .
  ...
  vars A B C : Bool .
  eq true and A = A .
  eq false and A = false .
  eq A and A = A .
  ...
endfm
```

We omit some parts of specifications by the dots (...). In this case, we omit the declarations of operation symbols and their related equations of `_or_`, `not_`, etc. The declaration `protecting` M means that the module imports M with the protect mode. In `TRUTH`, the sort `Bool`, the constant `true` and `false`, the equality predicates and `if_then_else_fi` mentioned above are defined. If M' imports M, the contents of M are included in M'. The rank of the operation symbol `_and_` is `Bool Bool -> Bool`, which means that for terms t_1 and t_2 of `Bool`, t_1 `and` t_2 is also a term of `Bool`. In the square brackets, attributes of the operation symbol are declared. The attribute `assoc` means that `_and_` is associative, i.e. (t_1 `and` t_2) `and` t_3 = t_1 `and` (t_2 `and` t_3). We can avoid brackets and write t_1 `and` t_2 `and` t_3 without any ambiguous parsing. The attribute `comm` means that `_and_` is commutative, i.e. t_1 `and` t_2 = t_2 `and` t_1. The attribute `prec` n means that the precedence is n. Lesser precedences indicate stronger connectivity in term expressions. For example, the precedences of `_and_`, `_or_` and `not_` in `BOOL` are 55, 59 and 53 respectively. For example, the term `not true and false` parses as `(not true) and false`, and is equivalent to `false`. Equations are declared with `eq`. The first equation means that the term `true and` t is equivalent to t for each term t of `Bool`. The second equation means that the term `false and` t is equivalent to `false` for each term t of `Bool`. The last equation means that the term t `and` t is equivalent to t for each term t of `Bool`. The operation symbol `_and_` denotes a function satisfying those equations, that is, the logical conjunction.

We use another built-in module `INT`, which includes the sort `Int` and the constants ..., `-2`, `-1`, `0`, `1`, `2`, ..., and the operation symbols `_+_`, `_*_`, `_>_`, ... The following is an extension of `INT` by adding the constant `na`:

```
fmod INTex is
  extending INT .
  op na : -> Int .
endfm
```

In `INTex`, `INT` is imported with the extending mode, which allows us to add new elements. The constant `na` is an element of `Int` which is not equivalent to any integer.

[1] Kanazawa University
[2] Minia University, Egypt
[3] Japan Advanced Institute of Science and Technology
[4] The Maude System, URL: `http://maude.cs.uiuc.edu/`
[5] MPI Forum : `http://www.mpi-forum.org/`

3. Syntax of PPL

In this paper, we specify a simple parallel programming language which supports some message passing functions. We call it PPL. In this section, we give a syntax of PPL: expressions and programs.

3.1 Expressions

PPL supports only the integer as a primitive data type. In the built-in module `INT`, we can deal with integer expressions like `1 + 2 * 3` as a term of `Int`. In PPL, we deal with an expression which may involve variables, like `x + 1 * y`. For this purpose, we describe the following specification `EXP` of the syntax of PPL expressions:

```
fmod EXP is
  protecting VAR .
  protecting INTex .
  sorts Exp .
  subsort Var Int < Exp .
  ops (_+_) (_*_) (_-_) (_=_) (_>_) (_and_)
      (_or_): Exp Exp -> Exp [ditto] .
  op not _ : Exp -> Exp  .
endfm
```

An order on sorts are declared as `subsort Var Int < Exp`, which means that terms of either `Var` or `Int` are also terms of the sort `Exp`. For example, The term `x` of `Var` and the term `1 + 2 * 3` of `Int` are also the term of `Exp`. Operation symbols in `INT` and `BOOL` are declared in the module `EXP` again, but they are defined on the sort `Exp`. The attribute `ditto` means that if imported modules include the operation symbol whose name is same, the new one inherits the attribute from the previous one. Maude allows overriding of operation symbols. Thus, terms containing variables and integers can be terms of `Exp`. For example, `x + 1 * y` and `x > 1 and y > 3` are terms of `Exp`. Note that the integer 0 is used as false and non-zero integers are used as true in PPL.

3.2 Programs

A program of PPL is a sequence of variable declarations, assignments, conditionals, and iterations. The specification `Pgm` of the syntax of PPL programs is given as follows:

```
fmod PGM is
  protecting EXP .
  sorts BPgm Pgm .
  subsort BPgm < Pgm .
  op int_; : Var -> BPgm
  op _:=_; : Var Exp -> BPgm [prec 38] .
  op if_{_} : Exp Pgm -> BPgm
  op while_{_} : Exp Pgm -> BPgm
  op __ : Pgm Pgm -> Pgm [assoc prec 41] .
  op end : -> Pgm .
endfm
```

First four operation symbols are constructors of basic programs (`BPgm`): variable declarations, assignments, conditionals and iterations. A variable declaration of `x` is represented by the term `int x ;` of `Pgm`. The term `x := y + 1 ;` represents the assignment which assigns the value of `y + 1` to `x`. The term `if x > 1 {x := 0 ;}` represents the conditional whose condition is `x > 1` and body statement is `x := 0 ;`. The term `while x > 1 {x := x - 1 ;}` represents the iteration whose condition is `x > 1` and body statement is `x := x - 1 ;`.

The operations symbol `__` is a constructor of programs. The operations symbol `__` just indicates positions of arguments. Because of the attribute `assoc`, a sequence of basic programs can be treated as a program, like BP_1 BP_2 BP_3. The constant `end` is used for the marker of the end of programs. The following is an example of PPL programs, which computes $\sum_{n=1}^{1000} n$:

```
int i ;  int x ;
x := 0 ;  i := 1000 ;
while i > 0 {
  x := x + i ;
  i := i - 1 ;
}
```

Note that the Maude system can treat the above PPL program directly as a term of `Pgm` without any transformation.

PPL supports message passing between processes. Each process can send a message to another process. The following is the syntax of message passing in PPL:

```
fmod PPGM is
  protecting PGM .
  op send(_,_); : Var Exp -> BPgm  .
  op recv(_,_); : Var Exp -> BPgm .
  op any : -> Exp .
endfm
```

The program `send(`X`,`E`);` tries to send the message X (the value of the variable X) to the process whose ID is equivalent to (the value of) the expression E. The program `recv(`X`,`E`);` tries to receive the message from the process E and assign the message to the variable X. The expression `any` is used for an arbitrary process.

4. Semantics of PPL

In this section, we give semantics of PPL. The semantics is based on the notion of stores, which is a model of storage.

4.1 Store

A store is given as a set of pairs of variables and values associated them. A store represents a state (or a snapshot) of storage in program running. The module `STORE` is given as follow:

```
fmod STORE is
  protecting EXP .
  sort Store .
  op _::_ : Var Int -> Store .
  op init : -> Store .
  op __ : Store Store -> Store
          [assoc comm id: init] .
  ...
endfm
```

The pair of a variable X and an integer I is denoted by the term $(X$`::`$I)$. The constant `init` denotes the initial empty store. From the attributes `assoc` and `comm`, a sequence of pairs can be treated as a (multi) set of pairs, e.g. $(P_1\ P_2\ P_3) = (P_3\ P_2\ P_1)$. The operation attribute `id: init` means that `init` is an identity element, i.e. $(P$ `init`$) = P$.

The module `STORE` includes the following operation symbols (in the omitted part): `in?(`X`,`S`)` checks whether the variable X is included in the store S or not. `val(`X`,`S`)` returns the value associated to X in S. `update(`X`,`I`,`S`)` updates the value associated to X to the integer I. For example, the equation `eq update(X, I, ((X :: J) S)) = (X :: I) S` is included in `STORE`.

4.2 Expressions

The value of an expression is determined by the current store. For example, the value of the expression `x + y` is `3` when the store is `(x :: 1) (y :: 2)`, and is `-1` when the store is `(x :: 1) (y :: -2)`. The value of the expression E on the store S is denoted by S`[`E`]`. The semantics of the expressions is given by the following module `SEM-EXP`:

```
fmod SEM-EXP is
  protecting EXP .
  protecting STORE .
  op _[_] : Store Exp -> Int .
  vars A B C : Var .
  vars I J K : Int .
  vars E E1 E2 : Exp .
  var S : Store .
  eq S[A] = if in?(A, S) then val(A, S)
                           else na fi .
  eq S[I] = I .
  eq S[E1 + E2] = (S[E1]) + (S[E2])  .
  ...
  eq (S[E1 = E2]) = (if (S[E1] == S[E2])
                       then 1 else 0 fi) .
  ...
endfm
```

The equations define the value S`[`E`]` inductively on the structure of expressions E. The first equation `eq S[A] = if in?(A, S) then val(A, S) else na fi` means that for a variable X, S`[`X`]` is defined as the value associated to X in the store S if S includes X, otherwise the special value `na`. The second equation `S[I] = I` means that the value of an integer (as an expression) is the integer itself. The third equation defines the value of the expression E_1 `+` E_2 by the values of each arguments E_1 and E_2. Note that the operation symbol `_+_` in the left-hand side `S[E1 + E2]` is a constructor of expressions declared in `EXP`, and the operation symbol `_+_` in the right-hand side `(S[E1]) + (S[E2])` is an operation symbol declared in `INT`. For example, `S[1 * x + y]` is equivalent to `1 * S[x] + S[y]`.

4.3 Programs

Semantics of PPL programs are given by a Maude system module, which specifies a rewrite system modulo equations, In a system module, we can declare rewrite rules: `crl` L => R `if` C. A term T is rewritten into T' (module equations) if there exists a rewrite rule such that T has an instance L' of the left-hand side L and the corresponding instance of the condition part C is equivalent to true, then, T' is obtained by replacing the instance subterm L' with the corresponding instance R' of the right-hand side R. The condition part can be omitted like `rl` L => R.

Programs modify stores. For example, for the store `(x :: 1) (y :: 2)`, the store modified by the program `x := x + y ;` should be `(x :: 3) (y :: 2)`. The store obtained by applying the program P to the store S is denoted by $S\ P$. A sequence of basic programs BP_1 $BP_2 \cdots BP_n$ modifies a store S_0 as follows:

$$
\begin{array}{lll}
S_0\ BP_1\ BP_2 \cdots BP_n\ \texttt{end} & \Rightarrow & S_1\ BP_2 \cdots BP_n\ \texttt{end} \\
 & \Rightarrow & \cdots \\
 & \Rightarrow & S_{n-1}\ BP_n\ \texttt{end} \\
 & \Rightarrow & S_n\ \texttt{end} \\
 & \Rightarrow & S_n
\end{array}
$$

where each S_i is the store obtained by applying the basic program BP_i to the store S_{i-1}. Semantics of PPL programs (without message passing) is given by the following module `SEM-PGM`:

```
mod SEM-PGM is
  protecting SEM-EXP .
  protecting PGM .
  sort State .
  subsort Store < State .
  op __ : State Pgm -> State [ctor] .
  vars BP BP2 : BPgm . ...
   rl S end  => S .
  crl S (int A ;) =>  (A :: na) S
      if not in?(A,S) .
  crl S A := E ;  => update(A,S[E],S)
      if in?(A, S) .
  crl S if(E){P1} =>  S P1 if S[E] =/= 0 .
  crl S if(E){P1} =>  S    if S[E] == 0 .
  crl S (while E {P}) => S (P while E {P})
      if S[E] =/= 0 .
  crl S (while E {P}) => S
      if S[E] == 0 .
   rl S (BP P)  => (S BP) P .
   eq (S (BP P1)) P2 = (S BP) (P1 P2) .
endm
```

The first rewrite rule means that if the program reaches the end then the current store is returned as the final store.

Variable declarations The second rewrite rule defines variable declarations. The variable declaration `int A ;` updates the store `S` when the variable `A` is not included in `S`, i.e. `not in?(A,S)` is true. The updated store is `S (X :: na)` where `na` is the special integer constant denoting "not available".

Assignments The third rewrite rule defines assignments. The assignment `A := E ;` updates `S` when `A` is included in `S`. In the updated store, the value associated to `A` is the value `S[E]` of the expression `E` in the previous store `S`.

Conditionals The fourth and fifth rewrite rules define conditionals. The store obtained by applying the conditional `if(X){P}` to `S` is `S P` if the condition part holds, i.e. `S[T]` is true, otherwise, it is `S`.

The last equation is needed for the case that the sequence of basic programs in the body part of a conditional (or an iteration) is applied to the store. For such cases, only the top of the sequence is applied to the store, like `S if(E){BP P1} P2 => (S (BP P1)) P2 = (S BP) (P1 P2)`.

Iterations The sixth and seventh rewrite rules define iterations. The iteration `while(T){P}` applies `P` repeatedly until the condition part does not hold, i.e. `S[T] == 0`.

Sequences The last rewrite rule defines a sequence of basic programs. The top of basic programs is consumed first.

4.4 Execution

Maude specifications are executable. For a given system module, the Maude rewrite command `rewrite` takes a term and returns a term obtained by applying rewrite rules repeatedly until no rewrite rule can be applied to. A term which no rewrite rules can be applied is called a normal form. The following is an execution result of rewriting the term which represents the application of the PPL program shown in Section 3.2 to the initial store `init`:

```
Maude> rewrite init (
  int i ;  int x ;
  x := 0 ;  i := 1000 ;
  while i > 0 {
    x := x + i ;
    i := i - 1 ;
  }
end ) .
...
result Store: (i :: 0) x :: 500500
```

where `Maude>` is the prompt of Maude system. In the last line, `(i :: 0) (x :: 500500)` is returned as a normal form of the input term. As we expected, $\sum_{n=1}^{1000} n = 500500$ is associated to `x`.

4.5 Stores for parallel computing

Since plural processes run in parallel computing, a store should be assigned to each process. A set of stores represents a state of parallel computing in our model. The module `PSTORE` is given as follow:

```
fmod PSTORE is
 protecting STORE .
 sort PState .
 subsort State < PState .
 op nil : -> PState .
 op _|_ : PState PState -> PState
             [assoc comm prec 99] .
endfm
```

For example, when we run a program with three processes, a state (a snapshot) is represented by the term of the sort `PState` like

$$\begin{array}{ll} & (\texttt{pid :: 0})\ (\texttt{np :: 3})\ (\texttt{x :: 12})\ P_0 \\ | & (\texttt{pid :: 1})\ (\texttt{np :: 3})\ (\texttt{x :: 3})\ (\texttt{y :: 1})\ P_1 \\ | & (\texttt{pid :: 2})\ (\texttt{np :: 3})\ (\texttt{y :: 2})\ (\texttt{z :: 1})\ P_2 \end{array}$$

where `pid` and `np` are reserved variables (in PPL) representing a process ID and the number of all processes respectively, which we declared in `VAR`. P_i is the remaining program to be executed in the process i.

4.6 Message passing

The semantics of message passing functions `send` and `recv` is given by the following system module `SEM-PPGM`:

```
mod SEM-PPGM is
 protecting PPGM . protecting PSTORE .
 inc SEM-PGM .
 vars S1 S2 : Store . vars P1 P2 : Pgm .
 var L : PState .  vars Dest Source : Exp .
 vars X1 X2 : Var .

 crl ((S1 send(X1, Dest);) P1)
   | ((S2 recv(X2, Source);) P2)
 =>  (S1 P1)
   | (update(X2, S1[X1], S2) P2)
 if
    S1[Dest] == S2[pid]
    and S1[pid] == S2[Source] .

 crl ((S1 send(X1, Dest);) P1)
   | ((S2 recv(X2, any);) P2)
 =>  (S1 P1)
   | (update(X2, S1[X1], S2) P2)
 if
    S1[Dest] == S2[pid] .
endm
```

The first rewrite rule defines message passing with the functions `send(X1,Dest);` in a process and `recv(X2,Source);` in another process. The condition of the rewrite rule is that the destination of the `send` function (`S1[Dest]`) is equivalent to the process which tries to receive a message (`S2[pid]`) and the process which tries to send a message (`S1[pid]`) is equivalent to the source of the `recv` function (`S2[Source]`). If there exist functions `send` and `recv` which satisfy the condition, then both functions are consumed and the value associated to `X2` in the receiving process's store is updated by the value associated to `X1` in the sending process's store.

The sending function specified in PPL corresponds to the synchronous sending function `MPI_Ssend` in MPI, where the process sending a message should stop until the target process calls a matching receiving. The second rewrite rule also defines message passing with `send` and `recv` functions. The source of the receive function is set for an arbitrary source (`any`), and thus the message from any process can be received by `recv(`X_2`,any);`.

4.7 Initialization

For a given program P and a given natural number n which stands for the number of processes, we define the initial state as follows: `(pid :: 0) (np ::` n`)` P `| (pid :: 1) (np ::` n`)` P `|` $\cdots$ `| (pid ::` $n-1$`) (np ::` n`)` P. The following is the specification of the initialization of PPL:

```
mod PPL is
 inc SEM-PPGM .
 op run : Int Pgm -> PState .
 op run' : Int Pgm Int -> PState .
 vars I J : Int .
 var P : Pgm .
 eq run(I, P) = run'(I, P end, I) .
 ceq run'(I,P,J) = nil if J < 1 .
  eq run'(I,P,1) = ((pid :: 0) (np :: I)) P .
 ceq run'(I,P,J) = ((pid :: J-1) (np :: I)) P
                   | run'(I, P, J - 1)
     if J > 1 .
endm
```

We show two execution results of parallel programs with message passing:

```
Maude> rewrite
run(5,
  if(not(pid = 0)){
    send(pid,0);
  }
  if(pid = 0){
    int x ; int y ; int i ;
    y := 1 ;
    i := np ;
      while (i > 1) {
        i := i - 1 ;
        recv(x,i) ;
        y := x * y ;
      }
  }
```

```
) .
...
result PState:
  (pid :: 1) np :: 5 | (pid :: 2) np :: 5
| (pid :: 3) np :: 5 | (pid :: 4) np :: 5
| (i :: 1) (x :: 1) (y :: 24)
  (pid :: 0) np :: 5
```

In the first example, the input program can be seen from the third line to fifteenth line (`if(not(pid = 0)) ... y := x * y ;}}`). The input program consists of two blocks separated by the value of `pid`. The first half is for the processor whose `pid` is not zero and the latter half is for the process 0. In the first half, each process tries to send its ID to the process 0 (`send(pid,0);`). In the last half, for each $i \in \{1, \ldots, \mathtt{np} - 1\}$, the process 0 tries to receive the message `x` from the process i (`recv(x,`i`);`), and multiplies `y` by `x` when the receive succeeds (`y := y * x ;`), where `np` is the number of processes and the initial value of `y` is 1. Note that messages are received in order of decreasing process ID number.

In this execution, there are five processes to run the program in parallel (`run(5,...)`). The result (a normal form) can be seen in the last four lines. For readability, we modified the real output of Maude system by editing line breaks. In the above normal form, we can see that the final store of the process 0 is `(y :: 24) (x :: 1) (i :: 1) (pid :: 0) (np :: 5)`. The value of the variable `y` is 24 ($= 4 \times 3 \times 2 \times 1$) as we expected. The value of `x` is 1 since the messages have been received in decreasing order.

The following is the second execution result, where the input program is same as above except the source of the receiving function:

```
Maude> rewrite
run(5,
  if(not(pid = 0)){
    send(pid,0);
  }
  if(pid = 0){
    int x ; int y ; int i ;
    y := 1 ;
    i := np ;
      while (i > 1) {
        i := i - 1 ;
        recv(x,any) ;
        y := x * y ;
      }
  }
) .
...
result PState:
  (pid :: 1) np :: 5 | (pid :: 2) np :: 5
| (pid :: 3) np :: 5 | (pid :: 4) np :: 5
| (i :: 1) (x :: 4) (y :: 24)
  (pid :: 0) np :: 5
Maude> rewrite
```

Since the receiving function can receive the message from any source (`recv(x,any);`), the order of receiving is not fixed. Although in the former example above younger processes should wait to send their messages until older processes finish sending, in this example the message sending first can be received first. Thus, the latter one is improved in the view of running speed. The value of `y` is also 24. Note that the value of `x` is 4, which means that the last message has been sent from the processor 4 unlike the case of the program without `any` above.

In general, a term may have more than one subterm which rewrite rules can be applied to, and thus more than one normal form exist for a given term. For example, the normal form in the former example can be a normal form of the latter example. The rewrite command just returns one of the all possible normal forms.

5. Verification

One of the most important features of Maude system modules is that we can search all possible normal forms automatically. The following is the instruction of searching all normal forms of a given term t:

`search` t `=>!` *pattern* `such that` *condition* `.`

where *pattern* is a term which may have fresh variables and *condition* is a term of `Bool` which may involve the variables in *pattern*. Then, Maude system searches all normal forms which are instances of *pattern* and satisfy *condition*. The condition part can be omitted.

5.1 Indetermination

Since the execution result showed in Section 4.7 (the latter one) just shows one of the possible normal form of the input program, it does not guarantee that the value of the variable `y` always becomes 24. In order to verify that the value of `y` is always 24, we check all possible normal forms by the search command as follows:

```
Maude> search
run(5,
  ...
)
=>!
((pid :: 0) (y :: Y:Int) S:Store | L:PState)
such that (Y:Int =/= 24) .
...
No solution.
```

where *pattern* is `(pid :: 0) (y :: Y:Int) S:Store | L:PState` and *condition* is `Y:Int =/= 24`. The expression x`:`s is the variable x of the sort s. `Y`, `S` and `L` are fresh variables of the sort `Int`, `Store` and `PState` respectively. Therefore, the above execution tries to search a normal form whose value of the variable `y` in the process 0 is not 24. Maude system returns `no solution` (in the last line), which means that there are no such normal forms, that is, the value of `y` in the process 0 is guaranteed to be 24 in the final state of any possible parallel running.

Consider the program obtained by replacing the assignment `y := y * x ;` with `y := x - y ;`. The following is an execution result of the modified program:

```
rewrite run(5,   ...
  recv(x,any) ;
  y := x - y ;
  ... ) .
...
result PState:
  (pid :: 1) np :: 5 | (pid :: 2) np :: 5
| (pid :: 3) np :: 5 | (pid :: 4) np :: 5
| (i :: 1) (x :: 4) (y :: 3)
  (pid :: 0) np :: 5
```

The value of `y` in the process 0 is 3 ($= 4 - (3 - (2 - (1 - 1))))$).

Next, similar to the above search, we try to check whether the value of y in the process 0 is 3 in all normal forms or not.

```
search run(5, ...
  recv(x,any) ;
  y := x - y ;
  ...)
=>!
((pid :: 0) ... suchThat I =/= 3 .
```

Then, unlike the above program with `y := x * y ;`, Maude system returns ten solutions which satisfy the pattern and the condition. We show one of those ten solution as follows:

```
Solution 7 (state 67262)
...
L:PState -->
  (pid :: 1) np :: 5 | (pid :: 2) np :: 5
| (pid :: 3) np :: 5 | (pid :: 4) np :: 5
S:Store --> (i :: 1) (x :: 1) np :: 5
Y:Int --> -1
```

where Maude system shows the instance of all variables in the pattern. We can see that the value of y in this solution is -1 ($= 1-(2-(3-(4-1)))$).

5.2 Deadlock

Deadlock detection is another important task in verification of parallel programming. When the processes 0 and 1 try to send messages to each other, `send(x,0);` and `send(y,1);` should not be consumed and both processes cannot finish the remaining programs. Detecting such deadlocks is not easy task since the destination of a sending function may not be an integer but a variable. The value of a variable is changed while running the program. Thus, dynamic analysis is suitable for detecting deadlocks rather than static analysis.

Consider the following program:

```
if(not(pid = 0)){
  int x ;
  send(pid,0);
  recv(x,0);
}
if(pid = 0){
  int x ; int i ;
  i := 1 ;
    while (np > i) {
      recv(x,any) ;
      x := x * x ;
      send(x,i) ;
      i := i + 1 ;
    }
}
```

For each $i > 0$, the process i tries to send its process ID and if the sending succeeds then it receives a message from process 0. The process 0 tries to receive a message x from any source and then sends x^2 to the process i in order of $1, 2, \ldots, \mathtt{np}-1$. The following is the result of applying the rewrite command to the above program with five processes:

```
result PState:
  (x :: 1) (pid :: 1) np :: 5
| (x :: 4) (pid :: 2) np :: 5
| (x :: 9) (pid :: 3) np :: 5
| (x :: 16) (pid :: 4) np :: 5
| (i :: 0) (x :: 16) (pid :: 0) np :: 5
```

From this result, we cannot find any problem of this program. However, as we discussed above, the execution result just shows one of the possible normal forms, and it does not guarantee that the program is deadlock-free. To obtain deadlock-free programs, we need to check all possible executions by the search command. Now, we search all normal forms as follows:

`search run(5,...) =>! L:PState.`

Since the pattern is a single variable and there is no condition, Maude system returns all normal forms of the input term. We show one of those normal form as follows:

```
Solution 6 (state 20479)
...
L:PState -->
  (x :: 1) (pid :: 1) np :: 5
| (x :: 4) (pid :: 2) np :: 5
| ((x :: na) (pid :: 3) np :: 5)
   send(pid,0); recv(x,0); if pid = 0 ... end
| ((x :: na) (pid :: 4) np :: 5)
   recv(x,0); if pid = 0 ... end
| ((i :: 2) (x :: 16) (pid :: 0) np :: 5)
   send(x,np - i); i := i - 1 ; ... end
```

Although it is a normal form, programs remain in some processes. The processes 1 and 2 successfully consume all programs. The process 4 waits for a message from the process 0 (`recv(x,0);`), however the process 0 tries to send a message to the process 3 (`(i :: 2)`, `(np :: 5)` and `send(x,np - i);`). The process 3 also tries to send a message (`send(pid,0);`). Then, those processes are in deadlock.

If there is no normal form which involves remaining programs in the final stores of all processes, then the program is guaranteed to be deadlock-free since the Maude search command checks all possible normal forms.

5.3 Abstraction

To detect possible indetermination and/or deadlock, we need search all normal forms exhaustively. Since a purpose of parallel programs is to compute heavy tasks fast, verification of parallel programs should also be extremely heavy tasks. For speed up, we propose an abstraction of a part of the input program. We propose a way to introduce a function of programs in PPL as follows:

```
mod FUN is
  inc PPL .
  op funi : Int -> Int .
  op funp(_); : Var -> BPgm  .
  var S : Store .
  var X : Var .
  rl (S (funp(X);))
     => ((X :: funi(S[X])) S) .
endm
```

where the operation symbol `funi` is an abstract function on integers. Note that no definition of `funi` is included in the module. The abstract function `funi` can be considered as an arbitrary function. The term `funi(`n`)` represents the integer which the function `funi` returns for n. The operation symbol `funp(_);` is a program function in PPL which computes `funi`. The meaning of `funp(_);` is described as the rewrite rule, in which when the function `funp(X);` is called, the value of the variable `X` is updated by `funi(S[X])`. We show an execution result:

```
Maude> rewrite
run(5,
  if(not(pid = 0)){
    funp(pid);
    send(pid,0);
  }
  if(pid = 0){
    int x ; int y ; int i ;
    y := 1 ;
    i := np ;
      while (i > 1) {
        i := i - 1 ;
        recv(x,any) ;
        y := x * y ;
      }
  }
) .
...
result PState:
  (pid :: funi(1)) np :: 5
| (pid :: funi(2)) np :: 5
| (pid :: funi(3)) np :: 5
| (pid :: funi(4)) np :: 5
| (i :: 1) (x :: funi(4))
  (y :: 1 * funi(1) * funi(2) * funi(3)
                              * funi(4))
  (pid :: 0) np :: 5
```

In the input program, each process except 0 calls the function `funp(pid);` and send the value of `pid` to the process 0. The process 0 computes the multiplication of all received messages. Note that the value `1 * funi(1) * funi(2) * funi(3) * funi(4)` of `y` includes values returned by `funi`.

We can verify indetermination of the value of `y` in the process 0 although there is no definition of the function `funi`.

```
Maude> search
run(5,...)
=>!
((pid :: 0) (y :: Y:Int) S:Store | L:PState)
such that (Y:Int =/=
1 * funi(1) * funi(2) * funi(3) * funi(4)) .
...
No solution.
```

Maude system returns no solution, which means that in all normal forms, the values of `y` in the process 0 are equivalent. The reason why the verification succeeded is because the operation symbol `_*_` is declared as an associative and commutative operation symbol.

Consider the program obtained by replacing the assignment `y := x * y ;` with `y := x - y ;`. Then, the value of `y` in the process 0 is `funi(4) - (funi(3) - (funi(2) - (funi(1) - 1)))`. Consider the following search:

```
Maude> search
run(5,...)
=>!
((pid :: 0) (y :: Y:Int) S:Store | L:PState)
such that
(Y:Int =/=   funi(4) -
(funi(3) - (funi(2) - (funi(1) - 1)))) .
```

Maude system returns the twenty-three solutions. We show one of those solutions as follows:

```
Solution 23 (state 149473)
...
L:PState -->
  (pid :: funi(1)) np :: 5
| (pid :: funi(2)) np :: 5
| (pid :: funi(3)) np :: 5
| (pid :: funi(4)) np :: 5
S --> (i :: 1) (x :: funi(1)) np :: 5
Y:Int --> funi(1) - (funi(2) - (funi(3)
                  - (funi(4) - 1)))
```

Unfortunately, the result does not directly mean that the program is indeterminate with respect to the value of `y` since it depends on the definition of the function `funi`. For example, if `funi` is defined as `funi`$(n) = 0$ for all n, then `funi(4) - (funi(3) - (funi(2) - (funi(1) - 1)))` and `funi(1) - (funi(2) - (funi(3) - (funi(4) - 1)))` are equivalent to `1`. If we add the equation `eq funi(I) = 0` to the module `FUN`, then Maude system returns `no solution` for the latter search. When we use the abstract function and some solution is returned in indeterminacy verification, we need to check whether the solution is correct for the function under consideration.

6. Related work

In [4], an algebraic specification of imperative programs has been proposed by using the algebraic specification language OBJ3 [5]. Maude is a successor of OBJ3. The specification is based on a theory of storage. A variety of actual storage mechanisms satisfy it. In [9], a behavioral specification of imperative programming languages has been proposed by using the algebraic specification language CafeOBJ [2]. CafeOBJ is another successor of OBJ3. In behavioral specification terminology, the set of stores has been given as a hidden sort, and the behavior of programs has been described via behavioral operation symbols. Each actual storage mechanism which satisfies the behavior can be a model, that is, an implementation of the behavioral specification. In this paper, we give a more concrete model of storage in order to obtain an efficient way to verify parallel programs by exhaustive searching. The approaches of [4, 9] are suitable for interactive verification with the techniques of proof scores [3, 10]. Although our approach restricts a model of storage to the set of pairs of variables and their values, simulation is extremely faster than the approaches of [4, 9] and moreover we obtain fully-automatic verification based on exhaustive searching.

Our *PPL* can be considered as a simplification of MPI programs. Several methods and tools to verify MPI programs have been proposed (for example, [6, 8, 7, 12]). Because of the space constraints of this paper, we cannot refer to all methods and tools related to our study. Here, we refer to MPI-SPIN [12] [6] as one of the formal verification tools for MPI programs, which seems to be one of the most related approaches to us. MPI-SPIN is an extension of a famous model checker SPIN [7], and supports exhaustive searching for all possible execution paths like the Maude search command. There are several practical case studies [13, 14]. Although in our approach, all examples in this article are verified in seconds (on 2.66 GHz Intel Core 2 Duo, 4GB memory, MacBook Pro), those examples are small and not so practical. In MPI-SPIN, the user need to build a suitable model from MPI programs. In our

[6]URL: `http://vsl.cis.udel.edu/mpi-spin/`
[7]URL: `http://spinroot.com/`

study, as we showed, Maude system can deal with PPL program codes directly without any translation into other languages, and verification is done by manipulating program codes themselves. In any step of simulation and/or searching of a PPL program, a snapshot is represented by a pair of the current stores (variables and their values) and the remaining programs, which makes it easier to detect a problem of an input program, as we showed in Section 5.2. Thus, in the view of readability of simulation and/or verification, our approach has an advantage over other model-checking approaches which needs some transformation of MPI programs in languages (or models) supported by the model checker.

7. Conclusion

We proposed an algebraic specification of parallel programming language PPL, which supports message passing functions like those used in MPI programs, and showed that it is useful to verify properties particular to parallel programming, e.g. uniqueness of a value in all possible normal forms and deadlock-freeness, by using the exhaustive searching command supported by Maude system. To reduce state and time explosion of exhaustive searching, we proposed a way to abstract PPL programs by using an abstract function.

Maude system supports not only exhaustive searching but also LTL (linear temporal logic) model checking. We can verify not only invariant (or safety) properties, which ensures that something bad never happens, but also properties which can be written in linear temporal logic, for example, liveness properties, which ensures that something good eventually happens, and more complicated ones. To find case studies to show the usefulness of applying Maude LTL model checker to PPL is one of the future work. Our PPL supports only synchronous send and receive functions. There are other useful functions specified by MPI standard, for example, broadcast and reduce functions. To extend PPL by adding those functions is another future work.

Acknowledgment

This research has been supported by the Kayamori Foundation of Information Science Advancement.

References

[1] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[2] Razvan Diaconescu and Kokichi Futatsugi. *CafeOBJ Report*. World Scientific, 1998.

[3] Kokichi Futatsugi. Verifying specifications with proof scores in cafeobj. In *ASE*, pages 3–10. IEEE Computer Society, 2006.

[4] Joseph A. Goguen and Grant Malcolm. *Algebraic Semantics of Imperative Programs*. MIT Press, Cambridge, MA, USA, 1996.

[5] Joseph A. Goguen, T. Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. *Software Engineering with OBJ: Algebraic Specification in Action*, chapter Introducing OBJ*. Kluwers Academic Publishers, 2000.

[6] Bettina Krammer, Katrin Bidmon, Matthias S. Müller, and Michael M. Resch. Marmot: An mpi analysis and checking tool. In Gerhard R. Joubert, Wolfgang E. Nagel, Frans J. Peters, and Wolfgang V. Walter, editors, *PARCO*, volume 13 of *Advances in Parallel Computing*, pages 493–500. Elsevier, 2003.

[7] Guodong Li, Michael Delisi, Ganesh Gopalakrishnan, and Robert M. Kirby. Formal specification of the mpi-2.0 standard in tla+. In Siddhartha Chatterjee and Michael L. Scott, editors, *PPOPP*, pages 283–284. ACM, 2008.

[8] Glenn R. Luecke, Hua Chen, James Coyle, Jim Hoekstra, Marina Kraeva, and Yan Zou. Mpi-check: a tool for checking fortran 90 mpi programs. *Concurrency and Computation: Practice and Experience*, 15(2):93–100, 2003.

[9] Masaki Nakamura, Masahiro Watanabe, and Kokichi Futatsugi. A behavioral specification of imperative programming languages. *IEICE Transactions*, 89-A(6):1558–1565, 2006.

[10] Kazuhiro Ogata and Kokichi Futatsugi. Some tips on writing proof scores in the ots/cafeobj method. In Kokichi Futatsugi, Jean-Pierre Jouannaud, and José Meseguer, editors, *Essays Dedicated to Joseph A. Goguen*, volume 4060 of *Lecture Notes in Computer Science*, pages 596–615. Springer, 2006.

[11] Peter Pacheco. *Parallel Programming With MPI*. Morgan Kaufmann, 1996.

[12] Stephen F. Siegel. Verifying parallel programs with MPI-Spin. In Franck Cappello, Thomas Hérault, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 14th European PVM/MPI User's Group Meeting, Paris, France, September 30 - October 3, 2007, Proceedings*, volume 4757 of *Lecture Notes in Computer Science*, pages 13–14. Springer, 2007.

[13] Stephen F. Siegel, Anastasia Mironova, George S. Avrunin, and Lori A. Clarke. Using model checking with symbolic execution to verify parallel numerical programs. In Lori L. Pollock and Mauro Pezzé, editors, *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17–20, 2006*, pages 157–168. ACM, 2006.

[14] Stephen F. Siegel, Anastasia Mironova, George S. Avrunin, and Lori A. Clarke. Combining symbolic execution with model checking to verify parallel numerical programs. *ACM Transactions on Software Engineering and Methodology*, 17(2):Article 10, 1–34, 2008.