Student Research Training Program Report

Linjie Tong, Ke Xu, Jiarun Hu

March. 12, 2022

1 Introduction

Reflection algorithm refers to an kind of algorithms or programs which has the ability to modify its own behavior. And Reflective algorithm has been considered as a feature component of future programming language with increasing interest to self-adaptive systems, while common natural joint cannot be expressed by parametric polymorphism but can be easily expressed by reflection.A recent researches has addressed the question how to verify and formalise the expected properties of a reflective algorithm. And it provides an natural example of how specify genetic algorithm by means of an rASM. [2]

While genetic algorithm is an example of Adaptivity in computing systems, which is designed to imitate processes of crossover and variation of chromosomal genes in biological evolution. Another similar algorithm is Simulated annealing (SA). But they have some crucial differences. The common SA always iterates a single point, and it's not parallel. However, the parallel recombinative simulated annealing (PRSA) solve this problem. [1]

The previous research has shown how to implement Genetic algorithms by means of an rASMS with sysntax trees representations. It iteratively transform a population of computer programs into a new generation if programs applying various "genetic" operations.[2] This procedure is sequential, only representing by one signature tree. Therefore, in order to explore further properties of reflective algorithms, the aim of this article is to investigate the relationship between reflective algorithm and parallel recombinative simulated annealing. Similarly extending sequential ASM to sequential reflective algorithm, we will extend parallel ASM to parallel reflective algorithm, and use it to specify PRSA.

A simplified Parallel ASM thesis have been found refers to K.D's work, whose idea is to modify bounded exploration postulate in sequential ASM by allowing non-grounded comprehension terms.[3] With this thesis and tree algebra for reflective algorithm, we can formally express the parallel rASMs.

2 Parallel Reflective abstract state machines

3 Example: PRSA

Similar with genetic programming, parallel reflective algorithm provides a example which can be specified by parallel reflective algorithm. It set T to a high value, initialize the population and generate each new population from current population by applying mutation, crossover and Boltzmann trials. [1]. In order to specify it by reflective algorithm, its aim is to iteratively generate population programs by applying applying genetic and annealing operations. The algorithm is defined by following

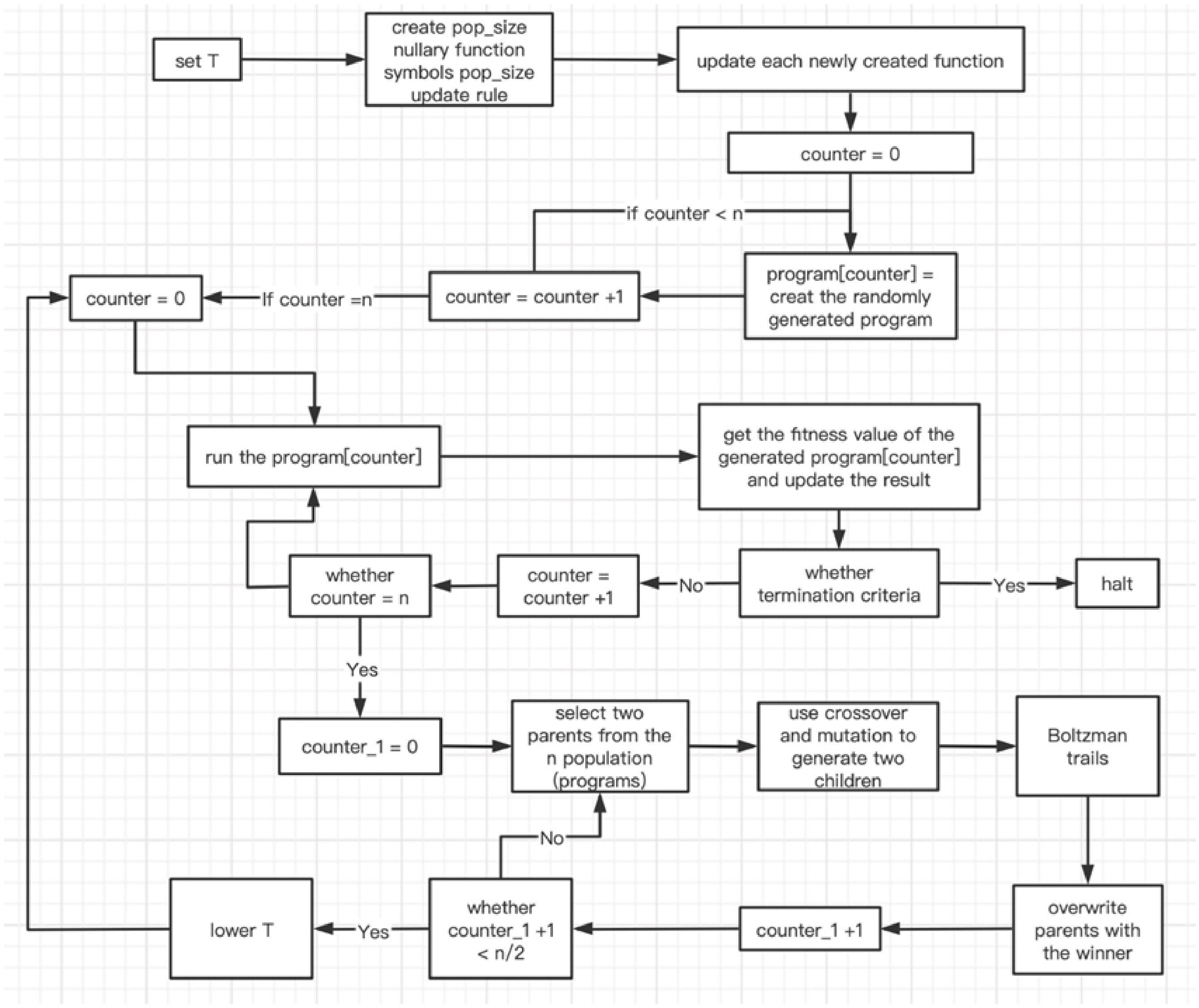
set T
create pop_size nullary function symbols pop_size update rule
update each newly created function
counter = 0
if counter < n
program[counter] = creat the randomly generated program
counter = counter +1
If counter =n
counter = 0
run the program[counter]
get the fitness value of the generated program[counter] and update the result
whether termination criteria
Yes
halt
No
counter = counter +1
whether counter = n
Yes
counter_1 = 0
select two parents from the n population (programs)
use crossover and mutation to generate two children
Boltzman trails
overwrite parents with the winner
counter_1 +1
whether counter_1 +1 < n/2
No
Yes
lower T

Figure 1: Flow chart of PRSA

Parallel recombinative simulated annealing

A) Set T (T is the temperature for Boltzmann Trial) to a sufficiently high value

B) creates *pop_size* new nullary function symbols as well as *pop_size* update rule which use to evaluate the new created function.

C) updated the newly created function with randomly generated program trees, and each of these trees is added to the initial generation of programs

D) Do n (n means population or the total number of program) times:

    Generate random program and named it as program[counter]

E) Do n times:

1) Run the program[counter]
2) Get the fitness value of program[counter]
3) Check whether the value meet the termination criteria.
    a) if meet, end the algorithm

b) if not, continue

F) Do n /2 times

  (a) Select two parents at random from the n population program
  (b) Generate two children using a recombination operator (such as crossover), followed by a neighborhood operator (such as mutation)
  (c) Run the two children program to get the fitness value of them
  (d) Do the Boltzmann trials between children and parents
  (e) Overwrite parents with the winner

G) Lower the T

H) Go to execute E

Just like in [2], we use following fixed background

- function $fitness$ for measuring the fitness of individual programs in the population
- A Boolean function *meet_term_crit* to judge whether the termination condition is satisfied
- A set T of well formed syntax trees of programs
- A function *prog* which maps each program tree in T to its corresponding expression
- A function $Boltzmax_dis$ which generate new offsprigng by *Boltzmann distribution*

Algorithm 1 Parallel Recombinative Simulated Annealing

```
1: ParallelRecombinativeSimulatedAnnealing($max\_d$, $init\_method$, $pop\_size$, $Temp$, $\Delta temp$) =
2: if $mode = init \wedge pop\_size < card(gen(0))$ then
3:     Import $x$ do
4:         ADDFUNC($x$)
5:         ADDUPDATERULE($x$)
6:         Seq
7:         GENRNDPROG($x$,$max\_d$, $init\_method$)
8: if $mode = init \wedge pop\_size = card(gen(0))$ then
9:     $n := 0$
10:    $mode := run$
11: if $mode = run$ then $mode := eval$
12: if $mode = eval \wedge \neg\exists x(x \in gen(n) \wedge meet\_term\_crit(result(x)))$ then
13:    mode:=reprod
14: if $mode = reprod \wedge pop\_size < card(gen(n+1))$ then
15:    Import $x$ do
16:        ADDFUNC($x$)
17:        ADDUPDATERULE($x$)
18:        Import $y$ do
19:            ADDFUNC($y$)
20:            ADDUPDATERULE($y$)
21:        Seq
22:        GENERATEOFFSPRING($x$, $y$, $max\_d$, $Temp$)
23: $Temp := Temp - \Delta Temp$
24: if $mode = reprod \wedge pop\_size \geq card(gen(n+1))$ then
25:    $n := n + 1$
26:    $mode := run$
```

The T is the temperature for Boltzmann Trial, and it should be set to a sufficiently high value initially. And the *Temp* is how much temperature will cool down for each generation. And the *Boltzmax_dis*() is used the Boltzman distribution to decide whether the Children or the parents will go to the next generation. Let's use *max_d* and *init_method* to represent maximum depth and method to create an initial random syntax tree of the program. Besides, *pop_size* is another important parameter about population size.

All these parameters will be updated in parallel at state with "mode = run". Each state position of the function symbol will be updated with a different random program tree and added to the initial generation of the program gen(0). And when "mode=eval" which means the evaluation mode, the algorithms will check whether need to terminate the progam. If the termination condition not met, mode will change to "reprod" which means that algorithms will go to reproduction mode and continue the previous process until termination.

Applying reflection, the *AddFunc* will add a new function symbol and *AddUpdateRule* will add an update rule to rASM. *GenRndProg* will specify the generation of program and *GenerateOffSpring* will specify the creation of offfsprings. And the priciple of "Choose by" is to find a program of current generation with probability based on fitness.

Algorithm 2 Generate a new offspring

```
1: GENERATEOFFSPRING($x$, $y$, $max\_d$, $Temp$) =
2: choose $z_1$ with $z_1 \in gen(n)$ do
3: choose $z_2$ with $z_2 \in gen(n) \wedge z_2 \neq z_1$ do
4:     choose $w_1$ with $w_1 \in nodes\_of(z_1)$do
5:     choose $w_2$ with $w_2 \in nodes\_of(z_2)$do
6:         $z_1' := subst_{tt}(z_1, w_1, subtree(w_2))$
7:         $z_2' := subst_{tt}(z_2, w_2, subtree(w_1))$
8:         choose $n_1$ with $n_1 \in nodes\_of(z_1')$do
9:         choose $w \in T$ with $depth(w) + level(z_1') \leq max\_d$ do
10:            let $z_1' := subst_{tt}(z_1', n_1, w)$
11:        choose $n_2$ with $n_2 \in nodes\_of(z_2')$do
12:        choose $w \in T$ with $depth(w) + level(z_2') \leq max\_d$ do
13:            let $z_2' := subst_{tt}(z_2', n_2, w)$
14:        $(x, y) := Boltzmax\_dis(result(z_1), result(z_2), result(z_1'), result(z_2'))$
```

Algorithm 3 Add a new function symbol of arity 0 to the signature

```
1: ADDFUNC($x$)= let $s = \mathbf{I}o.(root(self) \prec_c o \wedge label(o) = signature)$ in
2:     $s \leftleftarrows^{right\_extend} label\_hedge(func, \langle x \rangle, \langle 0 \rangle)$
```

Algorithm 4 Add update rule to the body of the if-rule

```
1: ADDUPDATERULE($x$)=
2: let $r = \mathbf{I}o.\exists o_1 o_2 o_{31} o_{32} o_{33} o_4 (root(self) \prec_c o_1 \wedge label(o_1) = \text{rule} \wedge o_1 \prec_c o_2 \wedge o_2 \prec_c o_{31} \wedge \forall z(z \not\prec_s o_{31}) \wedge o_{31} \prec_s$
   $o_{32} \wedge o_{32} \prec_c o_{33} \wedge o_{33} \prec_c o_4 \wedge label(o_4) = \text{bool} \wedge o_4 \prec_s o)$ in
3: $r \leftleftarrows^{right\_extend} label_hedge(\text{update}, \text{func}\langle\text{result}\rangle\text{term}\langle\text{x}\rangle\text{term}\langle\text{fitness(prog(x))}\rangle\rangle)$
```

Algorithm 5 Generate Random Program

```
1: GENRNDPROG($x$, $max\_d$, $init\_method$)=
2: if $init\_method = \text{full}$ then
3:     choose $y \in T$ with $y \notin gen(0) \wedge depth(y) = max\_d \wedge full\_tree(y)$ do
4:         $cur\_gen(0, y)$true
5:         $x := y$
6: if $init\_method = \text{grow}$ then
7:     choose $y \in T$ with $y \notin gen(0) \wedge depth(y) \leq max\_d$do
8:         $cur\_gen(0, y)$true
9:         $x := y$
```

4 Specify all Genetic Algorithm by reflective algorithm

Due to the definition of reflective ASM, The extension to reflective ASMs requires to define a background structure that covers trees and operations on them, a dedicated variable self that takes as its value a tree representation of an ASM signature and rule, and the extension of rules by partial updates. And in genetic programming, the first part is to apply some algorithms to code the populations to the string and then apply genetic operation to modify these strings. After that, decoding these strings and find the fitness of these strings and find the children for next turn. In this process, the coding string will be modified. And the another definition of reflective is that the program can modify itself while running. Here it comes to a kind of tree called abstract syntax tree (AST). AST can represent all kinds of code for its node represents a construct, its leaves represent value or valuable and its tree structure can represent the grouping parentheses. So by combining the coding strings and process of decoding into a syntax tree and make it to be parts of tree in the background of reflective ASMs. Thus every GA can be represented by rASM. Below is an example: The goal is to maximize $f(x) = x^2$ in $[0, 7]$ and x is integer. It will be demonstrated by a GA with population 2. I will show it for one round of GA running. For GA in ASM, firstly, generate 2 random populations: $S1 =' 110'$, $S2 =' 001'$ In this example, only crossover is used, and then $S1 =' 111'$, $S2 =' 000'$. Here, the contain in string is changed. After decoding, the fitness of S1 is 49 and that for S2 is 0. By applying discrete probability distribution, the new children will be $S1 =' 111'$, $S2 =' 111'$ and go to a new round. In rASM, however, the string will be substituted by the syntax tree using the decoding rule. The decode of $S1$ is $1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$. The decode of $S2$ is $0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$. And the decode of $S1$ can be represented using syntax tree:

Program: $S_n[2] \times 2^2 + S_n[1] \times 2^1 + S_n[0] \times 2^0$

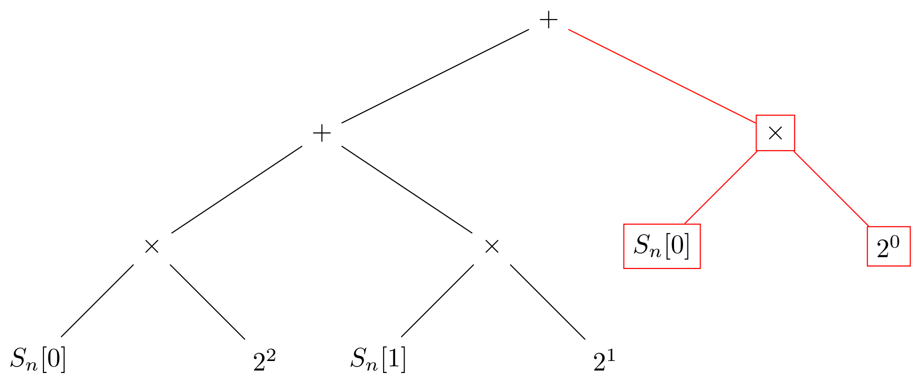

Figure 2: Tree of program population

Program of population 1: $S_0[2] \times 2^2 + S_0[1] \times 2^1 + S_0[0] \times 2^0 = 7$

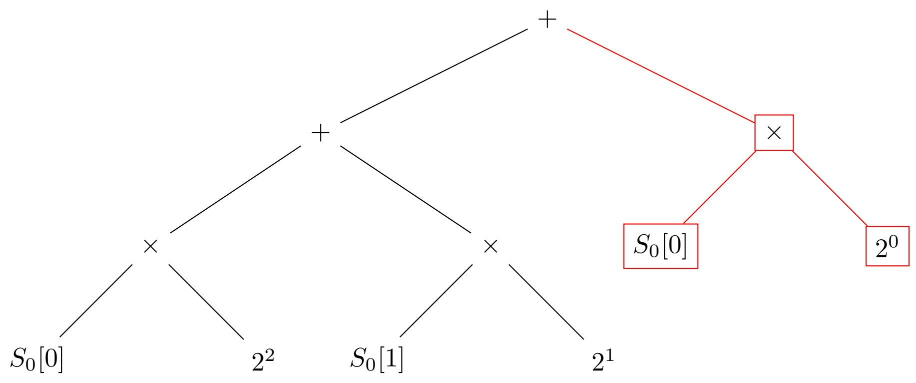

Figure 3: Population 1 of Example

Program of population 2: $S_1[2] \times 2^2 + S_1[1] \times 2^1 + S_1[0] \times 2^0 = 1$

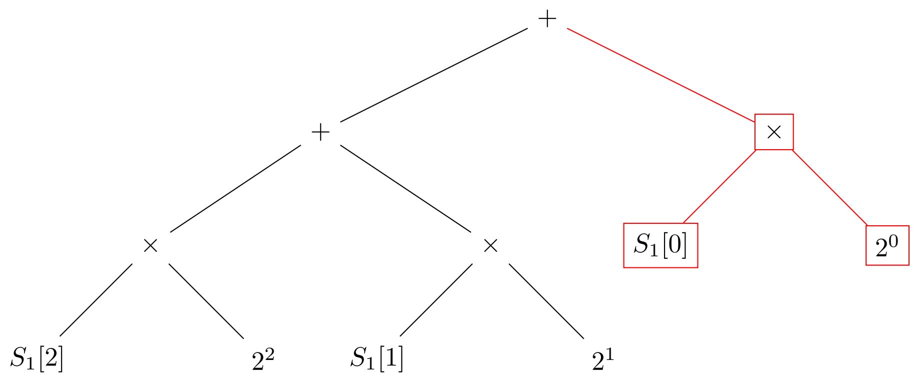

Figure 4: Population 2 of Example

The red parts is going to crossover. And after crossover the outcome is
New population 1 of program:$S_0[2] \times 2^2 + S_0[1] \times 2^1 + S_1[0] \times 2^0 = 7$

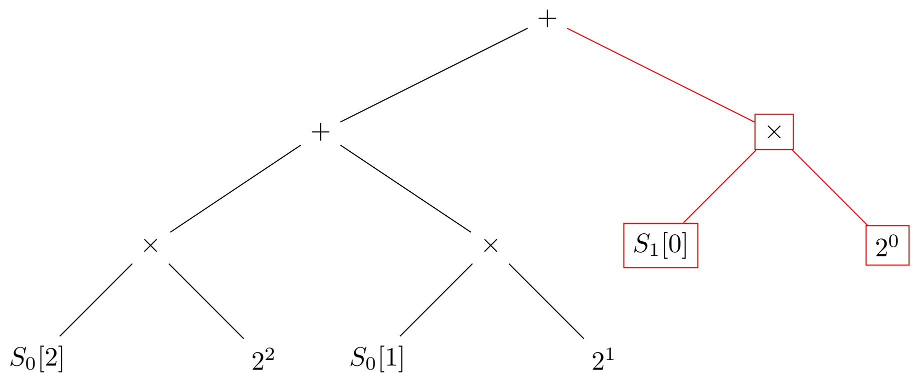

Figure 5: New Population 1 after crossover

New population 2 of program:$S_1[2] \times 2^2 + S_1[1] \times 2^1 + S_0[0] \times 2^0 = 0$

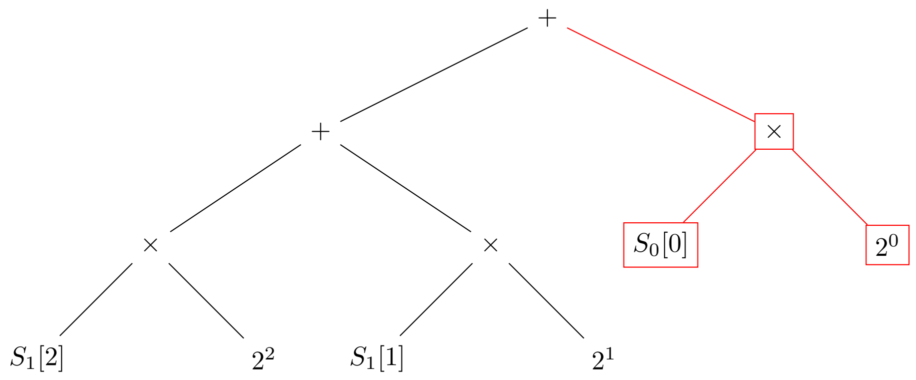

Figure 6: New Population 2 after crossover

References

[1] Samir W. Mahfoud and David E. Goldberg. “Parallel recombinative simulated annealing: A genetic algorithm”. In: *Parallel Computing* 21.1 (Jan. 1995), pp. 1–28. ISSN: 01678191. DOI: 10.1016/0167-8191(94)00071-H. URL: https://linkinghub.elsevier.com/retrieve/pii/016781919400071H (visited on 11/08/2021).

[2] Klaus-Dieter Schewe, Flavio Ferrarotti, and Senén González. “A logic for reflective ASMs”. In: *Science of Computer Programming* 210 (Oct. 2021), p. 102691. ISSN: 01676423. DOI: 10.1016/j.scico.2021.102691. URL: https://linkinghub.elsevier.com/retrieve/pii/S0167642321000848 (visited on 11/06/2021).

[3] Klaus-Dieter Schewe and Qing Wang. “A Simplified Parallel ASM Thesis”. In: *Abstract State Machines, Alloy, B, VDM, and Z*. Ed. by John Derrick et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 341–344. ISBN: 978-3-642-30885-7. DOI: 10.1007/978-3-642-30885-7_27.