Petra, A modification of Lindenmayer systems (L-systems) enabling modularity for applications in parallel and distributed software.

Aran Hakki, MEng Systems Engineering
University of Warwick
Founder of CognitionBox.io

14 January 2020, v1.3.1

Introduction

Lindenmayer Systems (L-system) were introduced in 1968 by Aristid Lindenmayer, a hungarian theoretical biologist. He formed the system to model the behaviour of plant growth and plant cell growth. The system is very simple, and from just a few simple rules, a huge variety of patterns can emerge. L-system's can be viewed as a parallel re-writing system and can model parallel software processes.

L-system Review

A L-system is a formal grammar consisting of 4 parts:

1. A set of variables: symbols that can be replaced according to the production rules (e.g. A, B, C, D …)
2. A set of constants: symbols that do not get replaced.
3. A single axiom which is a string composed of some number of variables and/or constants. The axiom is the initial state of the system. (e.g. ABCD).
4. A set of production rules defining the way variables can be replaced with combinations of constants and other variables. A production rule consists of two strings - the predecessor (P) and the successor (S). Production rules look for substrings within the system state that match it's predecessor string. Where a match exists, the match is replaced with the production rules successor string.

L-system Example

variables : A B
constants : none
axiom : A
rules : (A → AB), (B → A)
which produces:
$n = 0$: A
$n = 1$: AB
$n = 2$: ABA
$n = 3$: ABAAB
$n = 4$: ABAABABA
$n = 5$: ABAABABAABAAB
$n = 6$: ABAABABAABAABABAABABA
$n = 7$: ABAABABAABAABABAABABAABAABABAABAAB

If at any one time only a single production rule can be applied, then L-system is said to be deterministic. Deterministic L-systems, are of particular interest to modelling parallel software processes correctly. If multiple production rules can be applied to the same substring then one must be chosen with some probability, these L-systems are non-deterministic and are named stochastic L-systems.

The Petra L-system

A Petra L-system is a modified deterministic L-system.
A Petra L-system is a formal grammar consisting of 5 parts:

1. A set of variables: symbols that can be replaced according to the production rules (e.g. A, B, C, D …)

2. A single axiom which is a string composed of some number of variables. The axiom is the initial state of the system. (e.g. AABBCCDD)

3. A set of production rules defining the way variables can be replaced with combinations of variables. A production rule consists of two strings - the predecessor (P) and the successor (S). Production rules look for substrings within the system state that match it's predecessor string. Where a match exists, the match is replaced with the production rules successor string. Petra production rules can only contain one symbol for the predecessor string (e.g. A or B or ... etc). Petra production rules can take a predecessor and return no successor, hence effectively consuming the predecessor, e.g. (A →). Petra production rules must have a unique predecessors, so the rules will be order independent and therefore the system will be deterministic.

4. A set of join rules, similar to production rules but allow a predecessor string of any size as per original L-systems. Join rules run sequentially after the production rules. This is because join rules can specify a predecessor string which can contain strings from production rule predecessors, and predecessors from other join rules. This would make the system non-deterministic and therefore Petra L-systems require each join rule to run sequentially, in the order they are defined, after all the production rules have been applied. Join rules are needed for combining and aggregating results as Petra productions rules have been restricted to operate on individual symbols.

5. A single production rule which defines the overall behaviour of the Petra L-system (e.g. A → B). Each Petra production rule can be substituted with an entire Petra L-system. The axiom of this Petra L-system would be the predecessor of the substituted production rule, and its output would be a single variable that the system converges to. In order to converge, we need to reduce the system state string down towards a single symbol that matches the system's production rule successor. If the system does not converge to a single symbol it will not terminate and hence the overall behaviour can not be described by a simple production rule. Future versions could support non-termination for systems which cycle, via notation for steady states or expected states after a specified number of iterations. Such systems would include web servers or other servers which need to be up indefinitely. Having a simple high level definition of the desired behaviour allows a system to be easily substituted with an equivalent system and provides a simple specification for automated reachability checking. This helps to improves modularity, readability, regularization and safety.

The above differs from the original L-system by not having a notion of constants, restricting production rules to have a single predecessor symbol, allowing production rules to not return a successor, allowing production rules to be substituted with an entire Petra L-system, and through the addition of the join rules mechanism. This adds consumption of variables and modularity, to the original L-system.
We demonstrate the Petra L-system with examples 1 to 6 below.

Example 1

system rule: A → EF
variables : A B
axiom : A
production rules : (A → AB), (B → A)
join rules : (ABA → CD), (CD → EF)

which produces:
n = 0 : A
n = 1 : AB
n = 2 : EF

At each n we evaluate all the production rules in any order, then we evaluate the join rules in the order specified.

At n=1 production rules (A → AB) run and (B → A) cannot run, A is replaced with AB.

At n=2 both production rules (A → AB) and (B → A) run, AB is replaced with ABA, and join rule (ABA → CD) runs, replacing ABA with CD, allowing join rule (CD → EF) to run, replacing CD with EF.

Note if we flipped the order of the join rules the result would have ended at n=3: EF rather than n=2: EF, hence join rules are order dependant and must be evaluated sequentially in the order defined. We demonstrate this with an example below.

n = 0 : A
n = 1 : AB
n = 2 : CD
n = 3: EF

At n=1 production rules (A → AB) run and (B → A) cannot run, A is replaced with AB.

At n=2 both production rules (A → AB) and (B → A) run, AB is replaced with ABA, and join rule (CD → EF) is checked first, having no effect, then join rule (ABA → CD) is checked and runs replacing ABA with CD.

At n=3 both production rules (A → AB) and (B → A) do not run, then join rule (ABA → CD) is checked, having no effect, and finally join rule (CD → EF) is checked and runs first, replacing CD with EF.

Example 2

variables : A B
axiom : A
production rules : (A → AB), (B → A)
join rules : (ABA → A)

which produces:

n = 0 : A
n = 1 : AB
n = 2 : A
n = 3 : AB
n = 4 : A
and so on…

Here the join rule creates a cycle by replacing ABA with A.

Example 3

variables : A B
axiom : A
production rules : (A → B), (B → A)

which produces:

n = 0 : A
n = 1 : B
n = 2 : A
n = 3 : B
n = 4 : A
and so on…

Here in each even iteration of n (A → B) replaces A with B and (B → A) has no effect, however in each odd iteration of n (B → A) replaces B with A and (A → B) has no effect. Overall we have a cycle from A to B back to A and so on.

Example 4

variables : A B
axiom : A
production rules : (A → AB), (B →)

which produces:

n = 0 : A
n = 1 : AB
n = 2 : A
n = 3 : AB
n = 4 : A
and so on…

Here the production rules replace A with AB and then removes B cycling back to A.

Example 5

Let's consider X to be a production rule that replaces A with B.

X : A → B

Under the hood, X is an Petra L-system defined as:

variables : A B
axiom : A
production rules : (A → AB), (B → C)
join rules : (AC → B)

which produces:

n = 0 : A
n = 1 : AB
n = 2 : B

At n=1 A is replaced with AB.
At n=2 B is replaced with C, resulting in AC, and in this same iteration the join rules are evaluated directly after the production rules, resulting in AC being replaced with B.

Example 6

Let's consider Y to be a production rule that replaces A with B.

Y : A → D

Under the hood, Y is an L-system defined as

variables : A B
axiom : A
production rules : (A → B), (B → C), (C → D)

which produces:

n = 0 : A
n = 1 : B
n = 2 : C
n = 3 : D

For a Petra L-system to return a value it must converge to one variable it its set.
From this we can we see how both production rules X: A → B and Y: A → B can be implemented as an L-system. This means we can nest L-systems thus enabling the build-up of more complex L-system using smaller modules.

Petra L-System Summary

1. Production rules are modified to only allow one predecessor symbol and also to allow replacing matches with nothing, i.e. consume the match.
2. Join rules are added to allow for value aggregation.
3. L-systems can be nested within other L-systems, which is made possible by:
    a. Adding a termination condition which causes evaluation to stop when one variable in the L-system remains and this variable matches the L-system successor symbol.
    b. Allowing production rules to be substituted for entire L-systems.

This approach leads us towards a formal language that could make it easier to reason about large and complex parallel and/or distributed systems. If we replace Petra L-system symbols with “object-oriented” objects and the predecessors/successors with predicates on these objects, we can start to see how it maps into traditional programming techniques. Replacing the production and join rules with a variant of hoare triples e.g. replace P → Q with {P} {C} {Q}, where {C} can only mutate a portion of the heap governed by the same reference used to evaluate the predicate {P}, shows how we can perform practical computations whilst improving safety. Also in order to scale software systems in practice, states are built up from constituent states, so it makes sense to see if extensions to Petra L-Systems can support symbols which can be decomposed into constituent symbols. This would allow us to decompose large “object-oriented” objects into constituent objects that would become available for matching and processing in parallel, distributed environments.