Genesis Specification Document v.1.0

Alton Chiu (alton.chiu@utoronto.ca)

April 27, 2015

Contents

1 Introduction/Overview

We present *Genesis*, a program generation language to help address the need for simpler synthetic code generation. Genesis simplifies the expression of a set of generated programs and facilitates the generation of synthetic programs in a statistically controlled fashion. Genesis allows users to annotate a *template program* to identify code segments of the program they wish to vary. The user also describes the values each parameter in the code segments could take, and the desired statistical distribution of these values in a *Genesis program*. The Genesis preprocessor uses the annotations in the input template program and the information in the Genesis program to generate a user-specified number of *instance programs*. The values of each parameter in the generated instance programs is drawn from their corresponding distributions, thus statistically controlling the characteristics of these programs.

Genesis has functionality not present in other preprocessors or program generators, making Genesis unique in comparison. Genesis is more flexible than other preprocessors, as Genesis can generate multiple instance programs of varying lengths with different characteristics. This is done with the useful ability of Genesis to sample from declared numerical distributions, which in turn generates multiple instance programs from different samples from these distributions. Genesis can also generate multiple instance programs by enumerating distributions. As Genesis is a standalone preprocessor and not an extension a current program language, it is not limited to a generating programs for a single output language and can generate programs for multiple different output languages.

Genesis makes it easier on the programmer by removing the need for the user to write code handling the generation of the instance programs, which is handled automatically by the Genesis preprocessor. New testcases can be generated by making small changes to the annotations without a need for large overhauls, adding to the ease of use of Genesis. Genesis is the first tool that values greater ease at the cost of learning a new language as a trade-off. We ultimately believe that Genesis is a useful tool that eases the expression of large and diverse program sets, which may lead to better end results for domains such as software compiler testing and program auto-tuning using supervised machine learning.

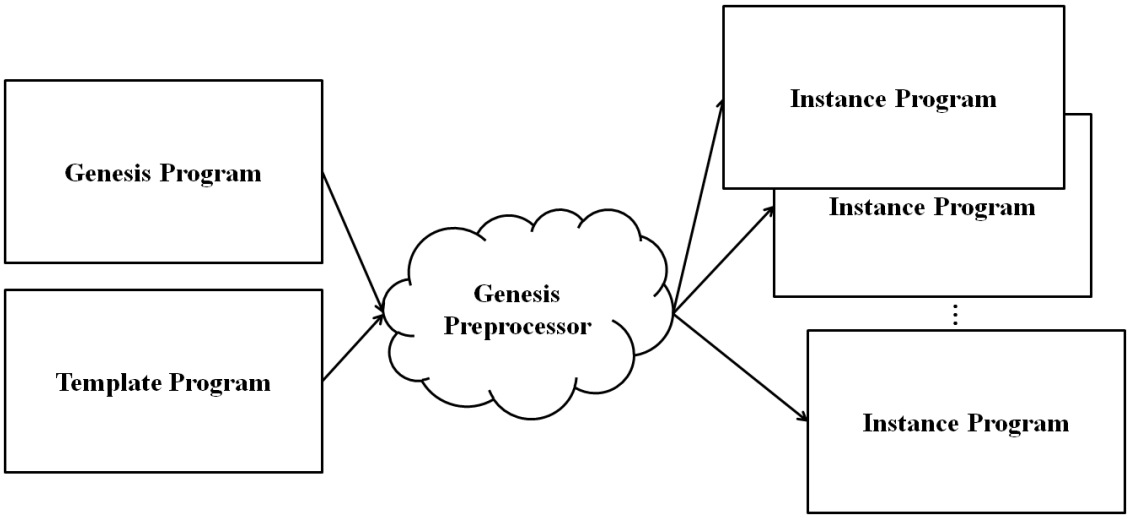

Figure 1: Overview of Genesis

Besides basic constructs that allow for the sampling of values and variables, more advanced constructs are available such as `genif` and `genloop` statements. Using too many conditional statements in the actual code results in a sub-par code quality. With `genif` statements, there is no conditional statements in the actual code and it is tailored based on the situation.

2 Model

Genesis defines a language whose constructs make it easy to generate synthetic programs. A user of Genesis starts with a *template program* in a target language of choice (C or C++, for example). The user identifies segments of this program that are to vary across a set of different *instance programs*, and replaces these segments with constructs of the Genesis language. The user also writes a *Genesis program* expressed in the Genesis language that describes how these segments are to be varied. The user then runs the Genesis *preprocessor* to process the Genesis and template programs, using these programs to generate a set of instance programs. These instance programs follow the form of the same template program, but are varied in the segments identified by the user. This high-level flow is depicted in Figure 1.
A simple example can be used to illustrate this flow and demonstrate some of the constructs of Genesis. Consider the following program, expressed in the C++ language:

```
#include <iostream>
using namespace std;
int main() {
    int x;
    cin >> x;
    cout << 2*x << endl;
    return (0);
```

```
}
```

In this program, the value of `x` is read from the standard input. This value is multiplied by 2 and the result is printed to the standard output. For this example, it is desired to generate a set of instance programs in which the integer multiplier of x is not always 2, but randomly varies from 1 to 10. Genesis can be used for this purpose.

The user of Genesis modifies the code above into a template program. A Genesis *feature reference* `${multiplyTerm}` replaces the term `2*x`, which is the part of the code which the user wishes to vary, as shown below.

```
#include <iostream>
using namespace std;
int main() {
    int x;
    cin >> x;
    cout << ${multiplyTerm} << endl;
    return (0);
}
```

This feature reference refers to a Genesis *feature* `multiplyTerm` in the Genesis program, which the user writes to define the code snippet and how it should vary. Thus, the user also writes a Genesis program containing this feature, as shown below.

```
1   begin genesis
2     distribution range = {1:10}
3
4     feature multiplyTerm
5       value coefficient sample range
6       ${coefficient}*x
7     end
8
9     generate 4
10
11  end genesis
```

The Genesis program uses the `begin genesis` and `end genesis` constructs on lines 1 and 11, respectively, to delimit Genesis code. Line 2 defines a Genesis *distribution* named `range`, which indicates the range of values from which the multiplier is to be randomly chosen, or *sampled*. The *feature definition* on lines 4 to 7 defines the `multiplyTerm` feature. It contains the declaration of a Genesis *value* named `coefficient` whose value will be randomly sampled from the distribution `range`. The code snippet of the multiply term itself is on line 6. This line uses the sampled value to `coefficient` using a *value reference*. The sampled value of `coefficient` will replace the reference, producing a final code snippet. Finally, line 9 uses the `generate` construct to indicate that 4 instances programs are to be generated.

The Genesis program and template program are then read by the Genesis preprocessor. It creates a copy of the template program to start generating an instance program. It scans the instance program for feature references and for each such reference, the corresponding feature defined in the Genesis program is *processed*. The preprocessor processes a feature by evaluating the Genesis code in its feature definition and producing

a resultant code snippet, called a *feature instance*. This feature instance *replaces* the feature reference in the template program. In this example, `${multiplyTerm}` is replaced by the code snippet returned by evaluating its feature definition. This produces a newly generated instance program with no remaining feature references.

```
1 #include <iostream>
2 using namespace std;
3 int main() {
4   int x;
5   cin >> x;
6   cout << 5*x << endl;
7   return (0);
8 }
```

```
1 #include <iostream>
2 using namespace std;
3 int main() {
4   int x;
5   cin >> x;
6   cout << 8*x << endl;
7   return (0);
8 }
```

Figure 2: Example Instance Programs Generated

This process repeats multiple times to generate multiple instance programs. Each instance program takes on the form of the template program, with a new processing of `multiplyTerm` replacing its reference each time. In each processing of `multiplyTerm`, the value `coefficient` is sampled again, taking on a new value ranging from 1 to 10. The result is a set of 4 instance programs that differ based on this sampling of `coefficient`. Figure 2 shows two possible generated instance programs.
Although this example is in C++, Genesis can work with different output languages, such as C (and its macro languages), Java and machine language. To keep Genesis language agnostic, it is up to the user to ensure that the instance programs are both correct syntactically (such that the instance program is valid) and semantically (such that the instance program runs correctly) in the target language.

2.1 The Flow of Genesis

As stated earlier, the preprocessor takes two inputs: a *template program*, expressed in a standard programming language, such as C, Java, or C++, and a *Genesis program*, expressed using the Genesis language.
The template program is a piece of code written in the standard programming language that each generated instance program will take its form of. The template program contains references to Genesis features to indicate the location of their intended use. These Genesis features are defined by the user in the Genesis program. The Genesis program defines Genesis *entities*, such as distributions, values, and features, also using code in the standard programming language mixed with Genesis code. The code in the standard programming language are written with references to other Genesis entities to indicate the locations of their use. These references are in the form of their given *Genesis name*, a user-given name to an entity, wrapped between `${` and `}`. For each instance program generated, value references will be replaced with their sampled values and feature references will be replaced by a processed feature instance. All feature references in the template program will be replaced by a differently processed feature instance, resulting in multiple varied instance programs.

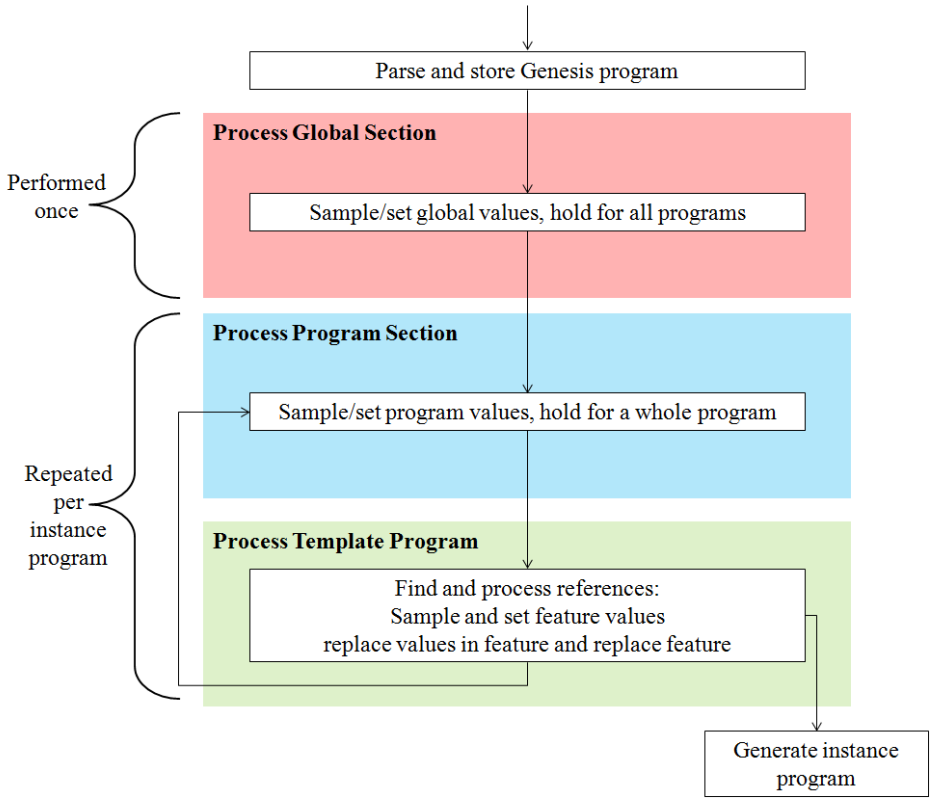

Figure 3: Flow of Genesis

Both the template program and features can contain references. The template program can only contain feature references. In contrast, feature definitions can contain feature references, as well as references to Genesis values and other Genesis entities. Genesis entities sample values at the point of their declaration and references to Genesis entities can only be used if that entity has already sampled a value. The exception to this are feature references, which processes a new feature instance once referenced.

Figure 3 depicts the flow of processing in Genesis. The preprocessor first reads the Genesis program to store its information. A Genesis program contains 3 sections: the *global* section, the *program* section, and the *feature definition* section. The preprocessor parses and stores all features that were defined in the feature definition section. This section contains the definitions of all features. Feature definitions are given a Genesis name and are processed once each time the feature is referenced.

Second, as depicted in the figure in the box labelled “Process Global Section”, the preprocessor processes the global section of the Genesis program, processing each statement sequentially. The global and program sections both contain Genesis values that define and perform samplings from distributions. The global section contains Genesis entities which are sampled once for the entire set of generated instance programs. These sampled values are stored for use in all instance programs.

Next, as depicted in the box labelled “Process Program Section”, the preprocessor processes the program section sequentially, where the values defined in this section are sampled. The preprocessor reprocesses the program section once for every instance

program. These sampled values are stored for use only for that instance program, with each entity sampling new values for each instance program.
Thus, the two sections differ by having different processing rates and by storing sampled values for different lengths of time. This controls the number of programs a sampled value should be held constant for. The entities defined in these two sections can be referenced in any feature. However, entities defined within a feature exist only in that feature and cannot be used outside that feature. Genesis entities with the same Genesis name can be defined in different areas of the Genesis program, but two Genesis entities with the same name cannot exist with a sampled value at the same point of time. An example demonstrating different processing rates is shown in Section **??**.
Then, as depicted in the box labelled "Process Template Program", the preprocessor then begins to generate an instance program by copying the template program. The preprocessor then searches for any feature references in the template program and, if any are found, processes the corresponding feature definition. The preprocessor replaces Genesis references in the definition by their sampled values or processed feature instance. This produces an actual code snippet that replaces the feature reference in the template program. A feature is generally processed once each time the feature appears in the template program. This processing repeats for every feature reference, creating a final instance program in the target language with no more feature references.
The preprocessor repeats the last two steps repeatedly to generate the indicated number of instance programs written in the `generate` statement. Each instance program at first is a copy of the template program. For each copy, the preprocessor re-samples the values in the program section, and processes the template program again with referenced features reprocessed. Processing all the copies results in the final, generated set of instance programs.

3 Running in the Command Line

Assuming that the Genesis preprocessor implementation genesis.pl is in the present working directory, in the general case, Genesis can be run by using:
`./genesis.pl [Genesis file]`
with the Genesis file containing both the Genesis program and the template program underneath. Perl may have to be explicitly declared, in which case `perl` is added in front of the previous command (or any other commands) as follows:
`perl ./genesis.pl [Genesis file]`
Alternatively, if the Genesis program and template program are in separate files:
`./genesis.pl [Genesis program] [template program]`
Inputting `./genesis.pl -h` outputs a usage line:

```
Usage:  ./genesis.pl [Genesis file] [options]
Usage:  ./genesis.pl [Genesis program] [template program] [options]

Possible options:
dataoutfile (can be Any Char-string)
outfile (can be string*string (ex.  Gen*.out))
```

Argument #	Values	Description
Genesis program	A string	The Genesis program to be parsed.
outfile	String*string Ex. gen*.out	The instance program format for instance programs
dataoutfile	A string	A place to put the parsing info if to the terminal.
finaloutfile	A string	The file for the final summary (not terminal).
outdir	A string	The directory for instance programs.
ignorevertspace	**0** 1	Does not ignore vertical spacing. Ignores blank lines. Use “genspace” to force.
generate	0 **1**	Parse Genesis program, do not generate. Parse Genesis program, generate instance programs.
printP	0 1 **2** 3	No info while parsing the Genesis program. Show a summary of parsed info at the end. Show info while parsing the Genesis program. Show more info while parsing the Genesis program.
printG	0 1 **2** 3	No info while generating instance programs. Show a summary of the instance programs at the end. Show info while generating instance programs. Show more info while generating instance programs.
globalcounters	**0** 1	Do not print the global counters. Print the global counters.
chisquared	0 **1**	Do not output the chi-squared test. Output the chi-squared test.
headercomments	0 **1**	Do not treat lines outside sections as comments. Treat lines outside sections as comments.
recursion	0 **1**	Do not allow recursion. Allow feature recursion.
header	0 **1**	Do not print a header in each instance program. Print a header in each instance program.

Table 1: Table of Command Line Arguments

```
outdir (can be Any Char-string)
ignorevertspace (default:  0.  1)
generate (default:  1.  0)
printP (default:  2.  1, 3, 0)
printG (default:  2.  1, 3, 0)
globalcounters (default:  0.  1)
chisquared (default:  0.  1)
header (default:  1.  0)
headercomments (default:  0.  1)
recursion (default:  1.  0)

Example usage:
./genesis.pl ./SpecFiles/Testcases/spec-demo.c printP=2 printG=2
```

Table 1 shows all the options available in the current preprocessor.

4 Terminology

This table describes some terminology that can be useful when reading through this document.

Term	Description
Genesis, or Genesis System	The collective name of the set of constructs used in the Genesis language and preprocessor described in this thesis used to increase the ease of generating synthetic programs
Genesis Preprocessor	The preprocessor that reads and parses Genesis programs to generate output synthetic programs
Genesis Program	A program written in the Genesis language that contain relevant Genesis information
Template Program	Code in a target language of choice containing Genesis references indicating parts that are to vary across a set of different *instance programs*
Instance Program	The generated instance created from a copy of the template program with the Genesis references processed and replaced.
Feature	Used to define code snippets that the user wishes to vary
Distribution	Used to define the range of values from which Genesis values is sampled from
Genesis Value	A Genesis entity whose value is randomly sampled from a distribution
Feature Reference	A Genesis construct indicating where processing and replacement of a feature should happen
Feature Instance	The resulted code snippet with different values and variables that is created from processing a feature section
Genesis Entity	A Genesis Object such as a Value, Variable, or Feature
Genesis Name	The given name to a Genesis Value, Variable, or Feature, used as a reference to indicate replacement
Processing	Creating a feature instance
Sampling	Assigning a proper value to a Genesis value/variable from a declared distribution
Replacing	Removing a Genesis name and placing the previously processed feature/sampled name in its place

Table 2: Terminology Table

5 Constructs

Section 5.1 gives a brief overview of the constructs that Genesis allows. This section goes in-depth on each construct.
When describing the format of each construct, non-italicised words are required when the construct is used, while *italicised* words provide options that are not required. **Bolded** words are constructs, written as shown, while non-bold words can be a string of alphanumeric characters and underscores or numbers, based on context explained in the section.

5.1 Constructs Overview

The following are a quick view list of constructs in Genesis.

`begin genesis`
Starts the genesis program.

`distribution`
Defines global distributions outside of the generate statement.

`value`
Defines a Genesis value that is sampled for each instance.

`varlist`
Defines a pool of variables, which Genesis variables can be sampled from.

`variable`
Defines a Genesis variable that is sampled for each instance.

`feature`
Defines a feature, defining a code snippet and how it varies.

`genif`
Defines an if statement that can generate code based on a sampled or defined value.

`genloop`
Defines a loop containing a iterator value that can be used in the loop.
Alternatively, Defines a loop with a condition.

`add`
Removes from a varlist.

`remove`
Adds to a varlist.

`genmath`
Evaluates the code in the brackets, or forces evaluation.

`genassert`
Asserts that a boolean expression is true. Ends the instance program if false.

`///`
Defines a comment that is not put into the actual code.

`generate`
Determines how many to generate and defines the distributions.

`end genesis`
Ends the Genesis program.

5.2 `begin genesis` Construct

This construct is needed at the beginning of the Genesis program and included in every Genesis program.

5.2.1 Format

`begin genesis`
The line is the same in every Genesis program.

5.3 `distribution` Construct

The **`distribution`** construct defines a set of values and their corresponding probabilities, storing these under a Genesis name. Distributions can either be defined using this statement or with the generate construct. Distributions are set once declared. This means that distributions declared using Genesis values will not change if the Genesis value is changed later on in processing.

5.3.1 Format

`distribution` `genesisName = distributionRange`
All the distributions are declared here.

`distributionRange` format is the following:
`{value`*`:endValue{distribution or increment};`*
`value:endValue{distribution or increment}... ;;distributionType``}`
Distribution percentage can be math. If distributions do not add to 100, the distributions are normalized, but the distributions still must be originally between 99 and 101 (or 0.99 and 1.01). Distributions can be a number (e.g. 0.5), a percentage (e.g. 50%) or a math expression (e.g. 1/2 or 4/2-3/2). Not including a distribution type defaults to uniform random. Currently, Genesis only allows uniform distributions, but other distribution types (e.g. normal distribution) can possibly be implemented in the future. Increment is better shown through an example (see examples dist4, dist5, and dist6).

5.3.2 Examples

`distribution` `dist0 = {1:10}`
Any Genesis value using `dist0` as its distribution has a 10% chance of sampling each value from 1 to 10.

`distribution` `dist1 = {0{1/4}, 1{1/8}, 2{0.375}, 3:10{1/4} }`
The chances of each value of being sampled are: 0 has a 1/4 chance, 1 has a 1/8 chance, 2 has a 3/8 chance, 4-10 have a 1/4, uniform for each value inside.

`distribution` `dist2 = {0, 1, 4:10;;uniform }`

0, 1 and 4-10 have a 33 chance each, individually, each value from 4-10 has a 33/7 chance each.

distribution dist3 = {0, 1, 4:10}
dist3 is same as dist2 as default distribution is uniform.

distribution dist4 = {2:32{*2}}
dist4 shows an increment in the brackets. Here the values in the distribution are 2, 4, 8, 16, 32, each with a 20% chance.

distribution dist5 = {2:32{+2}}
dist5 shows an increment in the brackets. Here the values in the distribution are 2, 4, 6, 8,...30, 32.

distribution dist6 = {040, 2:32{*2, 60}}
dist6 contains 0 with a 40% change, and {2,4,8,16,32} with a 60% chance (so 2 has a 12% chance, 4 has a 12% chance, etc.).

distribution dist7 = { 4:10;; real }
All real values from 4-10 can be sampled.

5.3.3 Restrictions

Using real assumes a single range only.
An error message occurs if more than 1 probability or increment is attached to a value. An error also occurs of the probabilities do not add up to 100, or a value is missing a probability when the other values in the distribution have probabilities.

5.4 `value` Construct

The **value** construct defines a Genesis entity whose value is sampled from a distribution. Values can be used with other constructs or can be propagated as a constant to the instance program. It can either take on a distribution, or copy the same value of a previously defined Genesis value.

5.4.1 Sampling Format

value genesisName **sample** takenDistName *modifiers*
This format is used for a value to be sampled from a distribution. genesisName can be any string of alphanumeric characters and underscores, starting with an alphanumeric character. takenDistName can be a string a characters, but should exist as a distribution declared earlier in a distribution statement, or later with the generate

statement. genesisName, when sampled, takes on a value based on the distribution takenDistName.

value genesisName **sample** {distribution} *modifiers*

A distribution can be declared in-line.

5.4.2 Enumerate Format

value genesisName **enumerate** takenDistName *modifiers*

This format is the same as above, but using the word enumerate instead of sample. Instead of sampling from a distribution, a Genesis value can also **enumerate** one. This causes **genesisName** to enumerate through all the values in **takenDistName**, taking on every possible value of **takenDistName**, one per instance program.

5.4.3 Math Format

value genesisName = ${takenGenesisName} *modifiers*

value genesisName = mathExpression *modifiers*

These formats are used when a previously sampled value, or a math expression is used to calculate the new value. genesisName can be any string of alphanumeric characters and underscores, starting with an alphanumeric character. takenGenesisName can be a string of characters, but should exist as a Genesis value declared previously. mathExpression is a math expression; all sampled values (if any) is replaced into the formula and the resultant value is stored as the value.

5.4.4 Initialization Format

value genesisName

This initializes a Genesis value, which can be assigned a sampled value later, using genmath. Its use without a value results in an error. This can be useful with multiple genif statements; initializing the value to set its declaration location, and using genif statements to assign it a value.

5.4.5 Other Formatting Details

Values declared over multiple lines can be compressed to a single line by separating declared values with commas:

value genesisName,genesisName2... **sample** takenDistName, genesisName3,genesisName4... **sample** takenDistName2, ...

Here, multiple values are declared using the same distribution, with multiple declarations on a single line. In general, these values are sampled with replacement.

Alternatively, arrays can be used for multiple sampling.

value genesisName[number] **sample** takenDistName

This results in number of genesisNames, referred in the code as genesisName[1], genesisName[2]... genesisName[number].

5.4.6 Modifiers

Modifiers can be added onto the end of lines, with default values used if not indicated. Values have a few adjustable modifiers. Setting an modifier twice results in an error.

report

With reporting set to 1 report(1), each time the value is sampled and used, a line is added at the beginning of each instance program indicating its name and its sampled value. **The header command line option has to be turned on.** The code results in the following at the top of each file:

```
//Sampled Reported Values:
//Set value numEpochs:  1
// Set value numComps:  5
//  Set variable dest:  temp5
```

noreplacement

With this modifier set to 1, multiple values sampled in the current line are sampled without replacement.

5.4.7 Examples

value stride **sample** dist1 report(1)

A Genesis value is declared called stride and it samples a value from a distribution called dist1. By using report(1), each time it is sampled, the result is reported as a comment at the beginning of each instance program.

value stride2 = ${stride}

A Genesis value is declared called stride2 and it has the same value sampled by stride.

value stride3 = ${stride}+1

A Genesis value is declared called stride3 and it has the same value sampled by stride plus 1. This value does not have to be part of the original distribution. If the distribution used by stride is 1:10, and stride is sampled to be 10, stride3 is 11.

value stride4 **enumerate** dist1

A Genesis value is declared called stride4, which enumerates through all values in dist1, one per instance program.

value stride5

A Genesis value is declared called stride5, without a given or sampled value. A value can be given later using genmath.

```
genmath stride6 sample {1:10}
```
`stride6` is sampled from the declared distribution.

5.4.8 Sampling without replacement

More than one value can be sampled at the same time using commas. By adding `without replacement`, the 2nd sampled value samples from the remaining variables not already sampled.

```
value swapIter,swapIter2 from swapIterDist without replacement
```
`swapIter` samples from `swapIterDist`. `swapIter2` samples from the remaining values in `swapIterDist`.

An error is given if the sampling results in an empty pool. For example, if 5 values are to be sampled without replacement when the distribution contains 4 numbers, this results in an error.
Arrays can be dynamic, as of Genesis 1.01.000. For example, the following code is now valid:

```
value test sample dist1 report(1)
value arrayEx[${test}] sample dist1 report(1)
```
A dynamic value array of size `test` is created.

5.4.9 Limitations

Enumerate is not properly implemented when used inside a feature. Enumerate only works on the regular level of the global and program section. Using `enumerate` constructs in these sections may not work as intended.

5.5 `varlist` Construct

The **`varlist`** construct defines a pool of variables that can be used in the processed feature and hence, be part of the instance program. The section in which the varlist is declared indicates the reinitialization rate of the varlist. If a varlist was manipulated using `add` or `remove` constructs, a varlist is *reinitialized* by undoing those actions, returning to a full varlist with all its variables. Varlists declared in the global section are created once for the entire set of programs. The size of the varlist and state of variables are maintained between instance programs in this case. Varlists declared in the program section are reinitialized at the point of its declaration, and thus, it return to a full varlist with all its variables for each instance program. Varlists declared in a feature are local to that feature only and are reinitialized for each processing of the feature.

5.5.1 Format

varlist varlistName [number] *modifiers*
varlist varlistName [genesisName] *modifiers*

varlistName can be any string of alphanumeric characters and underscores, starting with an alphanumeric character. number indicates a set number of variables, genesisName allows for a variable number of variables based on a Genesis value that was declared earlier.

5.5.2 Modifiers

Modifiers can be added onto the end of lines, with default values used if not indicated. Varlists have a few adjustable modifiers. Setting an modifier twice results in an error. Setting both an init and an initall causes an error.

type

Indicates a type for that varlist. float is the default type if none is indicated.

init

Gives the varlist a different initial value. Values should be separated with commas. If not indicated, the default is that it equals the previous value. Init values can be sampled values. Without this clause the default for the first is 0, and equal the previous one for the rest. For example:

```
value0 = 0;
value1 = value0;
value2 = value1;
...
```

This initializes all values to the previous value.

initall

This modifier gives all values in the varlist the same init value.

endtouch

endtouch(0) indicates that a generated line should touch all elements in the varlist at the end.

name

name allows the user to define a character string that goes before the number in the output variables. By default, if this modifier is not used, the name of the varlist is used as that string. For example, a varlist declared using temp[5], has variables ranging from temp1 to temp5. However, if it is declared using temp[5] name(foo), the variables are named from foo1 to foo5.

5.5.3 Examples

```
varlist vars[5]
```

A `varlist` called `vars` containing 5 variables is initialized, named from `vars1` to `vars5`.

```
varlist foo[${someValue}] name(bar)
```

A `varlist` called `foo` containing some variables, where the number of variables is `someValue`, a value that should be declared as a Genesis value, and sampled in each instance of the program. The variables in the list are named `bar1`, `bar2`, etc.

5.5.4 Varlist References

When a varlist is referenced, statistics of the varlist can be queried using arguments.

```
(size)
```

size returns the size of the current varlist. For example, declaring using `temp[5] name(foo)` and referencing using `${temp(size)}` outputs `5`, in the general case. Using adds and removes can affect its return value.

```
(name)
```

name returns the name used in the variable. For example, declaring using `temp[5] name(foo)` and referencing using `${temp(name)}` outputs `foo`.

```
[a number]
```

Referencing with a number between the square brackets accesses that variable in the varlist. For example, declaring using `temp[5] name(foo)` and referencing using `${temp[3]}` outputs `foo3`.

5.5.5 Varlist Reference Examples

```
varlist bar[10]
genif ${bar} > 5
end
genif ${bar} > 15
end
```

The first `genif` evaluates to true and the second evaluates to false. Using `add` and `remove` constructs can manipulate the evaluation of `bar`.

5.6 `variable` Construct

The **`variable`** construct defines a Genesis entity whose value is sampled from a varlist and is propagated as a variable name to the target program. It samples from a variable pool with uniform distribution, or takes on the same variable name as another Genesis variable.

Each Genesis variable keeps track of their pool of available variables that can be pooled. These lists can be modified using the add and remove constructs.

5.6.1 Format

variable variableName **from** takenvarlistName *modifiers*
variableName can be any string of alphanumeric characters and underscores, starting with an alphanumeric character. takenvarlistName can be a string a characters, but should exist as a varlist declared earlier in the code. variableName, when sampled, takes on a variable in that varlist.

variable variableName = takenVariableName *modifiers*
variableName can be any string of alphanumeric characters and underscores, starting with an alphanumeric character. takenVariableName can be a string a characters, but should exist as a variable declared previous.

5.6.2 Modifiers

Modifiers can be added onto the end of lines, with default values used if not indicated. Variables have a few adjustable modifiers. Setting an modifier twice results in an error.

report
With reporting set to 1 report(1), each time the value is sampled and used, a line is added at the beginning of each instance program indicating its name and its sampled value. **The header command line option has to be turned on.** The code results in the following at the top of each file:

```
//Sampled Reported Values:
//Set value numEpochs:  1
// Set value numComps:  5
//  Set variable dest:  temp5
```

5.6.3 Examples

variable tempVar from vars
Variable tempVar is sampled from a varlist vars.

variable tempVar2 = ${tempVar}
Variable tempVar2 takes on the same variable as tempVar.

variable tempVar3 = ${tempVar}+1
Variable tempVar3 takes on the variable value after tempVar. For example, if tempVar is var1, tempVar3 takes on the variable var2. This DOES wrap around, as variables depend on factors such as initialization, that values do not require.

5.7 `genmath` Construct

The **`genmath`** construct allows the re-evaluation of previously declared Genesis values.

5.7.1 Math Format

`genmath` `genesisName = someExpression`

These formats are used when a previously sampled value, or a math expression is used to calculate the new value. `genesisName` and `takenGenesisName` can be any string of alphanumeric characters and underscores, starting with an alphanumeric character, as long as the Genesis name exist as a Genesis value declared previously. `mathExpression` means that math can be used; all sampled values (if any) is replaced into the formula and the resultant value is stored as the value.

5.7.2 Sampling Format

`genmath` `genesisName` **`sample`** `takenDistName` *`modifiers`*

This format is used for a value to be sampled from a distribution, useful for when a value is declared but not yet sampled. `genesisName` can be any string of alphanumeric characters and underscores, starting with an alphanumeric character, as long as it exists as a Genesis value declared previously. `takenDistName` can be a string a characters, but should exist as a distribution declared earlier in a `distribution` statement, or later with the `generate` statement. `genesisName`, when sampled, takes on a value based on the distribution `takenDistName`.

`genmath` `genesisName` **`sample`** {`distribution`} *`modifiers`*

A distribution can be declared in-line.

5.7.3 Examples

`genmath` `stride` **`sample`** `dist1`

A Genesis value is declared called `stride` and it samples a value from a distribution called `dist1`.

`genmath` `stride2 = ${stride}+1`

A Genesis value previously declared called `stride3` takes the same value sampled by `stride` plus 1. This value does not have to be part of the original distribution. If the distribution used by `stride` is 1:10, and `stride` is sampled to be 10, `stride2` is 11. The idea is that `stride3` should not be held down by the restrictions of `stride`, since a distribution is not indicated. There may be an argument in a future update that may control this.

`genmath` `stride3 sample {1:10}`

stride3 is sampled from the declared distribution.

5.8 `add` Construct

The **`add`** construct adds a variable to a varlist or a value to a distribution to affect future samplings.

5.8.1 Format

`add` `variableName` **`to`** `destinationName`

`variableName` can be any string of alphanumeric characters and underscores, either the name of a value, variable, or a number. `destinationName` can be a string a characters, but should exist as a varlist or distribution declared earlier in the code. The value of `variableName` is added to the allowable sampled values of `destinationName`.

5.8.2 Examples

`add` `dest` **`to`** `vars`

Add the sampled value of `dest` into the `vars` varlist.

`add` `stride` **`to`** `bar`

Add the sampled value of `stride` into the `bar` varlist.

`add` `1` **`to`** `bar`

Add the first variable back into the `bar` pool.

`add` `1` **`to`** `dist1`

Add `1` into `dist1`. It is normalized with the rest of the values in the distribution.

5.9 `remove` Construct

The **`remove`** construct removes a variable to a varlist to affect future samplings, such that it can be used to prevent repeated samplings.

5.9.1 Format

`remove` `variableName` **`from`** `sourceName`

`variableName` can be any string of alphanumeric characters and underscores, either the name of a value, variable, or a number. `sourceName` can be a string a characters, but should exist as a varlist or distribution declared earlier in the code. The sampled value of `variableName` is removed from the allowable sampled values of `sourceName`.

5.9.2 Examples

```
remove dest to vars
```
Removes the sampled value of `dest` from the `vars` varlist.

```
remove stride to bar
```
Removes the sampled value of `stride` from the `bar` varlist.

```
remove 1 to bar
```
Remove the first variable from the `bar` pool.

```
remove 1 to dist1
```
Removes `1` from `dist1`, assuming it exists in the distribution. The remaining values in the distribution are normalized.

5.10 `feature` Construct

The **`feature`** construct defines a code snippet built up using Genesis names or possibly other features. A feature is defined inside its own block in the feature section of the Genesis program.

5.10.1 Format

```
feature modifiers featureName
  code
end
```
`featureName` can be any string of alphanumeric characters and underscores, starting with an alphanumeric character. Description can be a set of code in the output language or declared local values and variables, or even other features.

```
feature modifiers featureName(listOfArguments)
  code
end
```
Arguments can be used to pass values into the code. The code can use these arguments as if the arguments were Genesis values.

5.10.2 Modifiers

Modifiers can be added onto the end of lines, with default values used if not indicated. Features have a few settable modifiers.

```
singleline
```

Default value is 0. Setting it to 1 results in the resultant code snippet to be in a single line with no newlines, and thus, the resultant C line is in a single line. For example, it can be used if a feature contains a **genloop** iterating through a varlist.

5.10.3 Examples

```
feature access
  variable dest from temp
  value stride1 sample dist1
  value stride2 sample dist1
  value offset sample dist1
  ${dest} = arr[${stride1}*it00 + ${stride2}*it01 + ${offset}];
end
```

This declares a feature named `access`. Values and variables sampled from distributions and varlists, and are used in the code snippet.

```
feature computation
  variable dest from temp
  variable source1 from temp
  variable source2 from temp
  variable source3 from temp
  ${dest} = ${source1} * ${source2} + ${source3};
end
```

A `computation` is generated and the code snippet replaces that feature reference.

```
feature epoch
  value numComps sample compDist
  ${computation[${numComps}]}
  ${access}
end
```

Example of using other features in the code snippet. Here, features are built using other features.

```
feature access2 (stride1,stride2,offset)
  variable dest from temp
  ${dest} = arr[${stride1}*it00 + ${stride2}*it01 + ${offset}];
```

end This example takes 3 arguments used in the code itself. See feature usage for examples on how to use this.

5.10.4 Restrictions

Feature names cannot contain take the same name as another Genesis entity.

Values declare in a feature can only be used in that feature. For use in a sub-feature, pass it as a argument.

5.11 `feature` Usage

A feature is used in the template program or in another feature by its name surrounded by curly brackets after a dollar sign, either in the template program or in other feature definitions. For each feature reference, the feature is processed and the resultant code snippet is substituted into the feature reference.

Features can also have arguments passed in by value.

5.11.1 Format

```
featureName (arguments) [number or genesisName]
```

A feature is referenced feature name as declared in the program.

5.11.2 Optional Clauses

Brackets allows arguments passed in by value, as required by the definition. Using square brackets allow for repetition. Inside the bracket can be a number or a Genesis value.

5.11.3 Examples

```
${access}
```

A single access replaced at the location of the reference.

```
${computation[5]}
```

5 differently sampled `computations` replaced at the location of the reference.

```
value numEpochs sample {1-5}
${epoch[${numEpochs}]}
```

Generate `numEpochs` number of epochs to replace this reference.

```
${access2(1,2,3)}
```

Here, `access2` is used with 1, 2, and 3 passed as arguments. Using the above example of the definition of `access2`, the arguments correspond to `stride1`, `stride2` and `offset` respectively.

```
${access2(1,2,3)[5]}
```

This examples shows the ordering of arguments and square brackets when used together.

5.12 `Stored features`

Processed features can also be stored for later referencing by a given Genesis name. A reference with the given Genesis name is substituted by the already processed code without reprocessing, similar to a Genesis value or variable.

5.12.1 Format

`feature` `featureName` **`process`** `takenFeatureName` *`modifiers`*

`featureName` can be any string of alphanumeric characters and underscores, either the name of a value, variable, or a number. `takenFeatureName` can be a string a characters, but should exist as a feature declared in the code. An instance of `takenFeatureName` is processed and stored into `featureName`. It can then be used and replaced by the same previously processed feature instance.

5.12.2 Examples

`feature` `stored_computation` **`process`** `computation`

A stored feature is declared called `stored_computation` as a stored processing of `computation`. The feature is processed during this line, and then can be used as many times as needed.

5.13 `genif` Construct

The **`genif`** construct is used for conditional generation of code snippets.

5.13.1 Format

```
genif booleanExpression
  description1
end
```

If `booleanExpression` is determined to be true, then `description1` is processed and placed into the instance program. Otherwise, this section of the Genesis program produces no code.

5.13.2 Optional Clauses

Using **`genelsif`** constructs after a `genif` statement allow for a second condition block that is only evaluated if the first **`genif`** statement is evaluated to false. **`genelse`** constructs allow a code section to be processed if all preceding **`genif`** and **`genelsif`** statements were evaluated to be false.

```
genif booleanExpression
  description1
genelsif booleanExpression2
  description2
genelsif booleanExpression3
  description3
...
genelse
  description4
end
```

If `booleanExpression` is determined to be true, then `description1` is processed and placed into the instance program. Otherwise, it continues checking each boolean expression until it finds a true expression, in which that description is processed. In no true expression exists, it processes the `genelse` clause. Even if no `genelse` clause exists, the end of the block needs to include the construct `end`.

5.13.3 Examples

```
genif ${ifChoice}==1
  code
end
genif ${ifChoice}==2
  otherCode
end
```

`ifChoice` is a Genesis value. If it is sampled as 1, code is used. If it is sampled as 2, otherCode is used. Any other value and neither are used (this part remains blank in the actual instance). Both sections are checked; the `genif` statements are independent of each other.

```
genif ${ifChoice}==1
  code
genelsif ${ifChoice}==2
  otherCode
genelse
  moreOtherCode
end
```

`ifChoice` is a Genesis value. If it is sampled as 1, code is used. If it is sampled as 2, otherCode is used. Any other value and neither are used (this part remains blank in the actual instance). Note here that if one section is evaluated to be true, further sections are not evaluated.

5.14 `genloop` Construct

The **`genloop`** construct facilitates repetitive code generation. The `genloop` construct can either be a counter format based on an increasing iterator, or can also test boolean conditions (similar to a while loop).

5.14.1 Counter Format

```
genloop genesisName:valueStart:valueEnd:valueStride
  description
end
```

The description is repeated `valueEnd - valueStart + 1` times. `genesisName` can be used as a Genesis value, which takes on the values from `valueStart` to `valueEnd`, and thus, can be used in the code.

5.14.2 Optional Clauses

A `valueStride` can be indicated. A stride of 1 is default.

5.14.3 Conditional Format

```
genloop condition
  description
end
```

This format is to use a `genloop` based on a condition (similar to a while loop). It is used with varlist with add/remove used.

5.14.4 Examples

```
genloop loopVar:1:5
  ${access(${loopVar})}
end
```

`loopVar` can be used like a Genesis value. It takes on the value of 1, and can be used in the code. Then, in the next iteration of the genloop, it takes on the value of 2, and can be used in the code. In this example, the code is repeated 5 times. `access`, assuming it was defined with an argument, takes the `loopVar` as an argument, taking on a different value in each iteration.

```
genloop loopVar:1:5:2
  ${access(${loopVar})}
end
```

Here, this line is the same as above, but the stride is 2. Thus, 3 accesses are made, where 1, 3 and 5 are passed in.

5.15 `genassert` Construct

The **`genassert`** construct makes an assertion of a boolean expression, similar to a **genif** statement. If the expression is evaluated to be true, processing of the instance program continues normally. However, if it is false, processing stops for the current instance program. Instead, it is considered a failed program and is deleted. The preprocessor then continues processing the next instance program. For example,

5.15.1 Format

`genassert` `booleanExpression`
It has a similar format to a `genif` statement. If booleanExpression is evaluated to be true, then processing continues normally. If it is false, processing stops for the current instance program, and processing continues for the next instance program.

5.15.2 Examples

`genassert` `${ifChoice}==1`
`ifChoice` is a Genesis value. If it is a value and sampled as 1, the generated instance program is valid at this point. If it is not 1, then the assumption is that something is wrong, and the current instance program has failed.

5.16 `geneval` Construct

This evaluates the code in the bracket and placed in the instance program after being evaluated.

5.16.1 Format

`geneval``(toBeEvaluated)`
Here, toBeEvaluated is any evaluable string (i.e., a string containing numbers or the characters + - * /), and is evaluated before being substituted into the instance program.

5.16.2 Examples

`2+1` is translated directly into the instance programs, without being evaluated.
`geneval(2+1)` results in 3 being put in the instance programs.

5.17 `generate` Construct

The **`generate`** construct defines how many instance programs to generate. The generate construct also allows the definition of global distributions.

5.17.1 Format

```
generate number
```
`number` is an integer to determine how many to generate.

```
generate number with distribution,distribution,...
```
`number` is an integer to determine how many to generate. Distributions not yet declared can be declared here too, in the same format as distributions declared using the `distribution` construct.

5.17.2 Examples

```
generate 500
```
Most basic generate statement with no distributions declared.

```
generate 500 with dist1 = {0{1/4}, 1{1/8}, 2{0.375}, 3:10{1/4} }
```
The chances of each value of being sampled are: 0 has a 1/4 chance, 1 has a 1/8 chance, 2 has a 3/8 chance, 4-10 have a 1/4, uniform for each value inside.

```
generate 100 with dist2 = {0; 1; 4:10;; uniform}
```
0, 1 and 4-10 have a 33 chance each, individual 4-10 have a 33/7 chance each.

```
generate 200 with dist3 = {0; 1; 4:10} , dist4={1;2}
```
`dist3` is same as `dist2` as default distribution is uniform. This example also shows multiple distributions being declared at once.

```
generate 100 with dist5 = { 4:10;; real }
```
All real values from 4-10 can be sampled.

```
generate 100 with dist6 = { 1:16{*2}}
```
`dist6` shows an increment in the brackets. The values of powers of 2 from 1-16 (1,2,4,8,16) are in the distribution.

Multiple generate lines in the same Genesis program is allowed.
```
generate 100 with dist7 = {4:10}
generate 100 with dist7 = {14:20}
```
This generates 200 total instance programs, 100 with each generate line.

Generate can be done over multiple lines, with an `end`. This keeps code cleaner.
```
generate 100 with
  dist8 = {4:10}
  dist9 = {0;1;15}
end
```

5.17.3 Restrictions

Using real assumes a single range only.

5.18 `end genesis` Construct

This construct is needed at the end of the Genesis program. The line is the same in every Genesis program.

5.18.1 Format

```
end genesis
```

The line is the same in every Genesis program.

6 Error and Warning Messages

Table 3 contains a list of errors that can appear while processing with the Genesis Perl preprocessor.

Code #	Message
”A”	Invalid value line.
”B”	Number to be distributed invalid.
”C”	Invalid varlist line.
”D”	Invalid variable line.
”E/E1”	Invalid add line.
”F/F1”	Invalid rem line.
”G”	Feature repetition value/number not valid.
”H”	Distribution for this value does not exist.
”I”	Varlist for this variable does not exist.
”J”	Distribution range end less than start value.
”K”	Varlist line argument invalid.
”L”	Feature line argument invalid.
”M”	Weird distribution value. Right chars, but possibly wrong format.
”N”	Invalid feature line.
”O”	Weird distribution value. Some char not allowed.
”P”	Varlist line argument structure invalid.
”Q”	Nothing in the brackets for value array.
”R”	Number for value array is invalid.
”S”	Too many arguments in this feature.
”T”	Too few arguments in this feature.
”U”	Genmath string not valid.

Continued on next page

Code #	Message
”V”	RHS or expression not fully evaluated for Genmath.
”W”	LHS does not exist to genmath.
”X/X2/X3”	Duplicate name exists as an Value/Variable.
”Y”	Not enough distribution to sample all without replacement.
”Z”	Value line argument invalid.
”AA”	Value line argument structure invalid.
”AB”	Missing ’end’ statement.
”AC”	No end genesis line.
”AD”	Way too many duplicates for some reason.
”AE”	Generate line in feature.
”AF”	Dynamic Not enough distribution to sample all without replacement.
”AG”	valueStart not fully evaluated.
”AH”	valueEnd not fully evaluated.
”AI”	valueStride not fully evaluated.
”AJ”	Genelsif without genif.
”AK”	Genelse without genif.
”AL”	Enumerate is invalid while in a feature.
”AM”	Type argument more than once.
”AN”	Init argument more than once.
”AO”	Initall argument more than once.
”AP”	Endtouch argument more than once.
”AQ”	Report argument more than once.
”AR”	Noreplacement argument more than once.
”AS”	Singleline argument more than once.
”AT”	Blank variable line.
”AU”	Start value not a valid value to sample from.
”AV”	End value not a valid value to sample from.
”AW”	Bad genif condition.
”AX”	Bad genelsif condition.
”AY”	Genelse should not have a condition.
”AZ”	Varlists is invalid (for now) while in a feature.
”BA”	Variable line argument invalid.
”BB”	Variable line argument structure invalid.
”BC”	Report argument more than once.
”BD”	Varlists using ’from’ should not have any arguments.
”BE”	Weird argument for generate.
”BF”	Value in the brackets does not exist.
”BG”	End is less than start of range.
”BH”	Feature repetition value not allowed in template program.
”BI”	Underscores allowed but cannot be first char.
”BJ”	Probability is non-numeric.
”BK”	Start value not defined yet.
”BL”	End value not defined yet.

Continued on next page

Code #	Message
”BM”	More than one program section.
”BN”	Feature cannot be named program.
”BP”	Distribution used already.
”BQ”	Add cannot work with real values.
”BR”	Remove cannot work with real values.
”BS”	Invalid value line. (Maybe change ’from’ to ’sample’.)
”BT”	Not a valid varlist target for a variable.
”BU”	Too many probability arguments.
”BV”	Too many probabilities.
”BW”	Too many increments.
”BX”	No number to generate!
”BY”	Blank value line.
”BZ”	Genloop with no arguments!
”CA”	’Type’ argument value invalid.
”CB”	’Init’ argument value invalid.
”CC”	’Initall’ argument value invalid.
”CD”	’Endtouch’ argument value invalid.
”CE”	’Report’ argument value invalid.
”CF”	’Noreplacement’ argument value invalid.
”CG”	’Report’ argument value invalid.
”CH”	’Singleline’ argument value invalid.
”CI”	Blank stored line.
”CJ”	First non-whitespace line is not ’begin genesis’.
”CK”	A line is outside the global/program/features section.
”CL”	More than one global.
”CM”	Varlist was not initiated yet.
”CN”	Invalid stored line.
”CO”	feature has recursion. A calling A
”CP”	feature has circular recursion. A calling B calling A
”CQ”	Varlist was not initiated yet for add.
”CR”	Varlist was not initiated yet for remove.
”CS”	Varlist was not initiated yet for add all.
”CT”	Varlist was not initiated yet for remove all.
”CV”	Varlist needs an argument value.
”CX”	Genif with no arguments!
”CY”	Genelsif with no arguments!
”CZ”	Feature cannot be named global.
”DA”	Inconsistency. Either give all probabilities this distribution, or not at all.
”DB”	Inconsistency. Either give all probabilities this distribution, or not at all.
”DC”	Probabilities not between 99 and 101.
”DD”	Reference found, no feature has this Genesis name.
”DE”	String has characters. Put it around quotes or fix the reference.
”DF”	Probably an endless loop of replacement.

Continued on next page

Code #	Message
”DG”	Varlist source does not exist.
”DH1/DH2”	Reference found with a non-existent Genesis name.
”DI”	Not a proper argument for a varlist
”DJ”	Value exists but has no sampled value yet.
”DK/DK1”	Value exists but has no sampled value yet.
”DL”	Either infinite recursion or an error with the compiler.
”DM”	The LHS value does not exist.
”DN”	Distribution in genmath statement does not exist.
”DP”	No number used for array size declaration.
”DQ”	No number used for array value reference.
”DR”	Number used for non-array value.
”DS/DS2”	Array number out of bounds: over.
”DT”	Array number out of bounds: under (-1).
”DU”	No index brackets used for array value.
”DV”	Varlist’s Value arg not a number.
”DW”	Number for Varlist’s value arg is out of bounds: over.
”DX”	Number for Varlist’s value arg is out of bounds: under.
”DY”	Varlist’s value arg needs an index.

Table 3: Table of Errors

The Genesis Perl preprocessor also contain bad flags/warnings, as listed in Table 4. Some warnings, such as Warning A, lead to the removal of the instance program, while others are allowed and purely informational to the user.

Code #	Message
”WB”	”Value”, Sampled at end of feature.
”WC”	”Varlist”, Sampled at end of feature.
”WD”	”Variable”, Sampled at end of feature.
”WE”	”Distribution”, Sampled at end of feature.
”WF”	Empty Distribution.
”WG”	Distribution declared but not used.
”WH”	No main segment.
”WJ”	No global segment.
”WK”	Code snippet in global section will not be used.
”WL”	Code snippet in program section will not be used.

Table 4: Table of Warnings

7 Evaluation Tools

8 Logging

Our implementation provides logging information to the screen for the user. In the general case, logging info can be changed with:
`./genesis.pl [Genesis program] printP=[level] printG=[level]`
where level can range from 0 to 3, with 0 meaning no logging, and 3 being the most logging. `printP` corresponds to printing during the first parsing phase and `printG` corresponds to printing during the second generation phase.
In the default mode, during the parsing phase, it informs the user of the type of each parsed line, such as `Value`, `Variable` or `Function Definition`. At the end of this phase, a quick summary of the parsed information is printed. During the generation phase, it informs the user of which instance program it is currently generating. At the end of this phase, a summary of the instance programs is printed, and the user is informed of any failed instance programs and any warning messages.
Our implementation provides a verbose mode, by setting some command line arguments (such as `printParsing`) from its default middle value of 2 to a high value of 3. This verbose mode prints more detailed information to the screen. During the parsing, the values being set, such as the name of the Genesis value and its arguments, are displayed. During generation, each time a Genesis value or variable is sampled, the user is informed, as well as any feature reference found.
Setting these command line arguments to a low value of 1 lowers the outputting to just the final summary at the end of each phase.
Counter info is provided, keeping track of how many times each distribution value is sampled by a Genesis value line. In the general case, global Genesis Value counters info can be changed with:
`./genesis.pl [Genesis program] globalcounters=1`
where 1 means the option is turned on and the information is printed at the end.
Our implementation provides a system to inform the user when something goes wrong, including a set of errors and warnings. Errors result in processing ending immediately, such as an incorrect Genesis Program. Warnings result in the instance program continuing to be processed, but the user is warned during the warning and at the end of processing. A list of all possible Genesis errors and warnings is included in Appendix 6.

8.1 Chi Squared Test

In the general case, logging info can be changed with:
`./genesis.pl [Genesis program] chisquared=1`
Even with a declared distribution, a chance exists that the actual set of sampled values can deviate greatly from the declared distributions. Thus, Genesis has tools to help the user of Genesis determine how well the actual set of sampled values match its specified distribution. The preprocessor also provides an option to count how often each value in a distribution is sampled for a Genesis entity, and then display these counts across all

```
numComps\\
\ \ value 1: 2\\
\ \ value 2: 5\\
\ \ value 3: 1\\
\ \ value 4: 11\\
\ \ value 5: 6\\
\\
Chi Squared Test\\
for numComps\\
 ColA: Value\\
 ColB: Actual Count\\
 ColC: Expected Count\\
 ColD: ColB-ColC\\
 ColE: ColD Squared\\
 ColF: ColE/ColC\\
\\
\ \ \ \ \ \ \ A\ \ \ \ \ B\ \ \ \ \ C\ \ \ \ \ \ \ D\ \ \ \ \ \ E\ \ \ \ \ F\\
\ \ value 1:\ \ \ 2\ \ \ 5.00\ \ \ -3.00\ \ \ 9.00\ \ \ 1.80\\
\ \ value 2:\ \ \ 5\ \ \ 5.00\ \ \ 0.00\ \ \ 0.00\ \ \ 0.00\\
\ \ value 3:\ \ \ 1\ \ \ 5.00\ \ \ -4.00\ \ \ 16.00\ \ \ 3.20\\
\ \ value 4:\ \ \ 11\ \ \ 5.00\ \ \ 6.00\ \ \ 36.00\ \ \ 7.20\\
\ \ value 5:\ \ \ 6\ \ \ 5.00\ \ \ 1.00\ \ \ 1.00\ \ \ 0.20\\
Final Chi Squared Value: 12.4\\
\\
\\
 The chi-squared value is between an alpha value of 0.01 and 0.02. This means this
     distribution differs from the declared distribution by between 98% and 99%. This means it
      may still be sampled correctly, but a high chance of bias affecting the sampling exists
     .\\
\\
Test Summary\\
 For numEpochs | Chi2: 2.00 | Alpha: 0.73-0.74 | Differs from declared: 26\%-27\%\\
 For numVars | Chi2: 2.00 | Alpha: 0.73-0.74 | Differs from declared: 26\%-27\%\\
 For stride1 | Chi2: 11.40 | Alpha: 0.24-0.25 | Differs from declared: 75\%-76\%\\
 For stride2 | Chi2: 9.00 | Alpha: 0.43-0.44 | Differs from declared: 56\%-57\%\\
 For offset | Chi2: 9.00 | Alpha: 0.43-0.44 | Differs from declared: 56\%-57\%\\
 For numComps | Chi2: 12.40 | Alpha: 0.01-0.02 | Differs from declared: 98\%-99\%
```

Figure 4: Terminal Output of Chi-Squared Test

instance programs. Using these values, the preprocessor outputs an analysis of the sampled values using Pearson's chi-squared test, an indicator test of how well the sampled distribution differs from the declared distributions. One downside of the chi-squared test is that while the chi-squared test is good for a general indication, instances exist where it does not work well as a simple test. For example, like most tests, the chi-squared test can not work well when the number of samples is small. It also cannot work when the distribution is real or continuous.

The preprocessor keeps track of sampling counts for each Genesis value, keeping track of how many times each value in a distribution is sampled. These counts are used to calculate and display a Pearson's chi-squared test number for each Genesis entity. This number provides a statistical confidence in the likelihood that the set of sampled values for an entity came from its specified distribution for that entity [**?**]. This Pearson's chi-squared test is used as a method of evaluation for our implementation of Genesis.

Version #	Date	Note
0.1	Aug 14, 13	Initial draft with all sections
0.2	Aug 19, 13	Overview and model
0.25	Aug 20, 13	Expanded model with a diagram and high-level overview
0.3	Sept 3, 13	Added loops, arguments, bug fixes, Dead code, variable pool issues (add/remove), proper spacing for readability
0.31	Sept 4, 13	Added a small tutorial overview, various readability problems fixed
0.32	Sept 9, 13	types indicated, distributions are used, if statements are inequalities
0.33	Sept 10, 13	Added more basis and explanation to tutorial
0.4	Nov 22, 13	Revised all the new stuff
0.5	Jan 3, 14	Added all appendices and some design decisions. Revisions.
0.6	Feb 21, 14	Update arguments, added and cleaned up document
0.7	Apr 1, 14	Cleaned up document, moved to latex

Table 5: Version History Table

9 Version History