BDS-WPT-MAGE-1.1.02
Extension of libGE for Attribute Grammars

March 20, 2014

Abstract

This Work Package task report concerns with extension of libGE, a C++ library for Grammatical Evolution, to allow the use of Attribute Grammars. This task report builds on the recommendations made in the task report DS-WPT-MAGE-1.1.01 on the current state of libGE and what search engines and evaluators to consider for this extension. The task reports should report on the progress made in programming the extension.

1 About this document

1.1 Revision History

Ver #5

Author : R. Muhammad Atif Azad

- First internal release: version 0.26.01. See section 2.

Ver #4

Author : R. Muhammad Atif Azad

- Completed skeleton implementations of `AGSymbol`, `AGContext`, `AGLookUp` and `AGDerivationTree`.
- `AGContext` can be a complex class given how the user views as the context of a node in an AGDerivationTree. Only providing the parent node and siblings to the left side as the context in `AGContext`, leaving the user to extend this class to provide more fancy contexts.
- `AGLookup` is a class that returns a suitable `AGSymbol*`. It is best to recycle objects but the current example implementation does not implement recylcing.
- Decided against using `AGDerivationTree` *is-a* `Tree` relationship; instead, used a *has-a* relationship.
- Need to implement a tree traversal mechanism for `AGDerivationTree`.

Ver #3

Author : R. Muhammad Atif Azad

- Continuing work to implement Multi-Knapsack problem with Attribute Grammars.

Ver #3

Author : R. Muhammad Atif Azad

- Tidied up the s-lang fitness function for a knapsack problem.
- Currently working towards adding functionality of adding problem specific extensions of `AGSymbol` class.
- Currently stuck at resolving a circular dependency between `AGSymbol` and `AGContext` classes. `AGContext` class defines the context of an attributed symbol so that the symbol in question can update its attributes by calling the appropriate symantic functions.

Ver #2

Author : R. Muhammad Atif Azad

- Figured out how to incprporate new classes in libGE. The process involves updating the file `Makefile.am` by incorporating the names of the class files and then running the command `autoreconf --install`. This generates a file `Makefile.in`. We can then run the `configure` script as usual and then `make` and `make install`.
- Added classes `src/AGSymbol.h, src/symbolLookUp.h` and `src/AGSymbol.cpp`.
- Read papers by Ryan and Karim to understand previous work with Attribute Grammars with the knapsack problems.
- Wrote a fitness function for the knapsack problems for s-lang.

Ver #1

Author : R. Muhammad Atif Azad

- Initial report.
- Agreed with Conor and James on the basic framework of classes to extend libGE. The crucial agreement is that the user will still specify the grammar in the traditional BNF notation but the semantic aspects will be specified in C++. To this end, each symbol containing attributes will be a C++ class that extends `AGSymbol`, another class that extends `Symbol` class from the current libGE framework. The semantic functions can be described either within these classes corresponding to the attributed symbols or in a separate C++ file.
- This report crucially depends upon other reports that suggest examples of Attribute Grammars.

2 libGE version 0.26.01: User Manual

Note: Currently sensible initialisation can NOT be used with Attribute Grammars. It should continue to work fine with the standard Context Free Grammars, though.

The subsequent sections expand on how to install, test run and program libGE0.26.01.

3 Platforms Tested

libGE version 0.26.01 has been tested to compile and execute error free on the following platforms:

- Mac OS X version 10.6.8; gcc version 4.2.1 (Apple Inc. build 5666) (dot 3).
- Ubuntu Release 13.04; version 4.7.3 (Ubuntu/Linaro 4.7.3-1ubuntu1).

3.1 Known Issues

libGE does not compile on Mac OS X version 10.9.2 (Maverick). This is not specific to version 0.26.01.

4 Installation

Note, this release has only been tested with MIT GAlib only (version 2.4.7) and s-lang (version 2.2.4).

The instllation instructions from the libGE manual apply; however, the following steps are recommended.

Although, the installation in the global directories should work, it is recommended to install everything relevant to libGE locally. Refer to libGE documentation to compare the benefits of local vs global installation,

Briefly, the following softwares should be installed:

- Consider a non-global installation (HINT: You should never have to run `sudo` while installing libGE, GAlib and s-lang (other evaluators).
- Install GAlib version 2.4.7: follow the instructions in INSTALL.unx file. Note, you will need to amend `makevars` file and set the `DESTDIR` variable to point to the location where you want GAlib to install.
- Install libGE. The configure script will have to be run with options appropriate to GAlib installation path. Refer to libGE documentation. For example, on the author's system, the following command was needed:

```
./configure
 --with-MITGALIBINCLUDES=/Users/azada/Documents/geproject/libGE/install/GAlib/
include/ga/
--with-MITGALIBLIBS=/Users/azada/Documents/geproject/libGE/install/GAlib/lib/libga.a
--enable-GE_ILLIGALSGA=no
```

Then run `make` and `make install prefix=libGEINSTALLPATH`. Replace `libGEINSTALLPATH` with a path of your choice.

- install s-lang version 2.2.4 as a shared library. The installation instructions are given in files INSTALL.*; e.g. use the file INSTALL.unx for all unix/linux based systems.

4.1 Test Run

Version 0.26.01 comes with an example problem, `Knapsack`, implemented. The implementation is in the directory: `EXAMPLES/AttributeGrammar/Knapsack/GE_MITGALIB/`.

In order to test run this example, first it should be compiled. The user will have to amend the `Makefile` in the aforementioned directory and set the appropriate paths for GAlib, libGE and s-lang. Currently, those paths are set as on the system of this author.

Then, run `make`.

After an error free compilation, run the following command:

```
./GESLANG popsize 500 ngen 50 grammar grammar.bnf seed 1 sensible 0 effective 1
```

This should lead to error free execution of GE with appropriate output. Output should also record in files `stats` and `is-output.dat`.

5 How to Program with libGE version 0.26.01

Version 0.26.01 extends libGE version 0.26 by enabling the use of Attribute Grammars. The subsequent sections desrcibe the nature of attribute dependency supported and the features enabling this dependency.

5.1 Attribute Dependency

This release supports both *inherited* and *synthesised* attributes. However, as depicted in Figure 1 some restrictions apply. In a derivation tree:

- the inherited attributes of a node can only depend on the inherited attributes of the parent node, and/or the inherited and synthesised nodes of the sibling nodes occurring on the left side.
- the synthesised attributes of a node can only depend on the synthesised attributes of the child nodes, and the inherited attributes of the node itself.

Such a dependency facilitates a depth first traversal of the derivation tree.

5.1.1 Constraints

The attributes can help impose syntactic constraints while deriving a tree. We can detect the violation of these constraints while computing the values of both the inherited and the synthesised attributes.

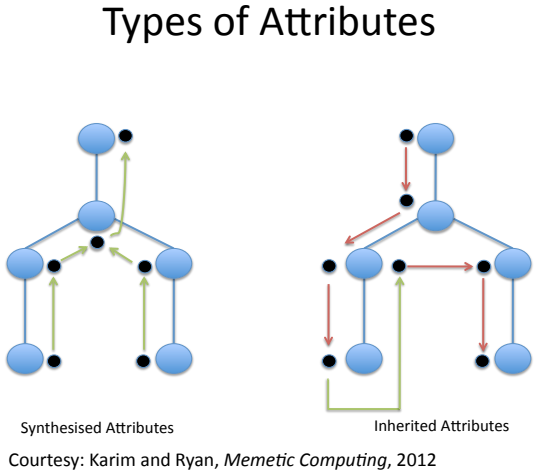

Figure 1: Dependency of inherited and synthesised attributes is shown. The red arrows correspond to inherited attributes, whereas the green arrows correspond to the synthesised attributes.

We can use two mechanisms to deal with constraint violations: these are *backtracking*, and *forward checking* (also called *look ahead*). If a chosen production violates a constraint, we can backtrack and choose a different production. However, when the problem specification allows, we can be proactive and pre-determine (or forward check) the productions valid in future. When practical, forward checking can save computational expense by avoiding restarts with backtracking.

5.2 Structure of Grammar is Critical to Problem Specific Code

It is crucial to finalise your grammar *before* writing code because the code will depend upon:

- the order of productions in the grammar.
- the order of symbols within a production.

.

If a change occurs in the grammar, the problem specific code should be re-examined for effects. For example, consider the following productions:

```
<K> ::= <I>
     | <I><K>
```

Let us call `<K>` on the left as `<K_1>` and that on the right side as `<K_2>`. Then, depending upon the values of attributes of `<K_1>` during

the mapping of an individual, the production `<I><K>` may be invalid. The production `<I><K>` is identified by its index. Therefore, changing the order of productions will affect the grammar specific code. You can of course write more sophisticated (and expensive to execute) identifiers for productions; however, in the absence of that, be careful in the ordering of productions.

The ordering of symbols within the production also matters because it defines how attributes in one symbol depend on those in another symbol. Section 5.3.3 explains this dependency in detail from an implementation point of view.

5.3 New Classes

The user should understand the role of the following classes before writing new code:

- `AGSymbol`;
- `AGLookUp`;
- `AGContext`

The knowledge of `AGDerivationTree` is useful but not necessary for the user.

5.3.1 AGSymbol

This class corresponds to every non-terminal and terminal symbol in the grammar. However, the user must subclass this class appropriately to provide both a suitable set of attributes and, functionality for updating those attributes for various symbols in the grammar. The implemented example (knapsack) contains extensions of `AGSymbol` in files `symbolclasses.h` and `symbolclasses.cpp`.

Note, that it may not be necessary to create a separate class for every symbol that occurs in the grammar file. For example, if a number of terminal symbols have the same data type, only a single subclass may be enough. `Item` class exemplifies this for all the terminal symbols in the Knapsack example.

The `AGSymbol` class declares two pure virtual methods that the user must over-ride:

- `updateSynthesisedAttributes(AGContext &, const int prodIndex, AGLookUp& lookUp);`
- `updateInheritedAttributes(AGContext &, const int prodIndex, AGLookUp& lookUp);`

which update the synthesised and inherited attributes respectively for the given AGSymbol object. Since their functionality is problem dependent, they are left as pure virtual methods.

The return type of both these methods is **AGMapState** which is an enum defined in `AGSymbol.h`. It has three possible values: `PASS`, `FAIL` and `CONSTRAINT_VIOLATION`. `PASS` signifies that updating the attributes did not violate any constraint and the mapping should continue normally. `CONSTRAINT_VIOLATION` represents a violation that should lead to *backtracking*. Normally, the implementations of these methods should return

one of only these two values (i.e. `PASS` or `CONSTRAINT_VIOLATION`) as appropriate.

The third value `FAIL` signifies a serious error that stops the mapping altogether for the tree under consideration. In this case backtracking can not happen; as a result, the tree stays incompletely mapped and is penalised with a poor fitness value. Therefore, it is not recommended to return this value unless the constraint violation is such that backtracking can not recover from.

To facilitate *forward checking*, where we can determine *in advance* which productions will not violate the given constraints, the class also declares the following virtual method with a default implementation.

```
const vector<bool>* getValidProductionIndices() const;
```

The default implementation only returns `NULL`. This means that functionality for forward checking is absent and the mapping will have to rely on backtracking alone to generate valid programs. However, to enable forward checking, the derived, problem specific classes should override this method. Often, this function should be called from within the `updateInheritedAttributes` method because the productions valid in future for this AGSymbol object should depend upon what this object is inheriting from its parent nodes. However, the user can also call it from the `updateSynthesisedAttributes` method if the problem so requires. The Knapsack problem exemplifies this.

Other methods that the user must implement in the derived classes of `AGSymbol` are:

- constructors;
- copy constructor;
- and the destructor.

Note that the derived classes can have any number and types of attributes as the problem requires. Although, the problem specific code can name the attributes as desired, it is intuitive to begin the names of synthesised attributes with `s_` (e.g. `s_usedItems`) and those of inherited attributes with `i_validIndices`. Again, the Knapsack example exemplifies that in `symbolclasses.h`.

5.3.2 AGLookUp

This is a class that implements the interface of a symbol look up table. When the BNF grammar is read in and the non-terminals and terminal symbols have to be matched with problem specific classes descending from AGSymbol, this interface is used.

Note, there is no default implementation of this class. The user **must** provide an implementation, as exemplified in the file `AGLookUp.cpp` in the Knapsack problem.

This class contains only two methods:

```
virtual AGSymbol* findAGSymbol(const Symbol& sym, const unsigned int productions =0,
bool grammarMade=false);

virtual ~AGLookUp();
```

This class is used for two purposes:

1. First, when constructing the grammar for the first time, each symbol that is read in must be matched with an AGSymbol object of *appropriate* derived class. At this time, the entire number of productions for that symbol may not have been read in, and the grammar is certainly not ready, so default values for the corresponding arguments are used as specified above in the function prototype.
2. Second, later when the grammar is ready and the individuals are being mapped, the first argument, Symbol object `sym`, can be downcast to one of the derived, problem specific, classes of `AGSymbol`. This is because `sym` has previously been constructed by this same method when the grammar was being first constructed. Again, Knapsack example shows how to do it.

5.3.3 AGContext

While expanding the derivation tree in the case of Attribute Grammars, attributes are both inherited (in a top down fashion) as well as synthesised (in a bottom up fashion). Thus, each node in the tree depends upon some other nodes which must be available when `AGSymbol::updateInheritedAttributes` and `AGSymbol::updateSynthesisedAttributes` methods are called. The `AGContext` class makes these nodes available.

Three methods are extremely important to understand to write problem specific code. These are:

1. `AGTree* getParent() const;`
2. `AGTree::iterator getStartIndex() const;`
3. `AGTree::iterator getEndIndex() const;`

`getParent()` method returns the parent node of the AGSymbol object under consideration. For example, if a production `<K_1> ::= <I><K_2>` has just been applied during mapping and update attribute methods for `<I>` are executing, then `getParent()` will return a pointer to the node that contains AGSymbol object containing `<K>`. Note, this will not return just *any* `<K>` but that which physically occupies the parent position in the derivation tree. Remember, subscripts 1 and 2 only differentiate between `<K>` on the left hand side and that on the right hand side.

`AGTree` is a type defined in `AGDerivationTree.h`. It is a tree where each node contains data of type `AGSymbol*`.

To understand the role of `getStartIndex()` and `getEndIndex()` consider the production `<K_1> ::= <I> <J> <K_2>`. Suppose, `updateInheritedAttributes` is called for `<K_2>`. The inherited attributes of `<K_2>` may depend on the attributes of it the parent node (`<K_1>` and also on the *sibling* nodes on the left hand side. Thus, `getStartIndex()` returns an iterator that points to the node containing `<I>`. This iterator can be incremented to get each subsequent sibling node *until* it reaches the value returned by `getEndIndex()`.

When called from `updateInheritedAttributes`, `getEndIndex()` points to the node containing `<K_2>`, therefore incrementing the iterator should

stop upopn reaching the value returned by `getEndIndex()` to iterate through siblings on the *left* side of `<K_2>`.

When called from `updateSynthesisedAttributes`, `getEndIndex()` points to the *end* of child nodes; that is, getEndIndex() does *not* point to a valid node, but it only marks the end point; thus, the iterator initialised with `getStartIndex()` should stop iterating when it equals the value returned by `getEndIndex()`. For example, if `updateSynthesisedAttributes` is called for `<K_1>`, then `getStartIndex()` points to the node containing `<I>`, whereas `getEndIndex()` points to the end past the node containing `<K_2>`. Thus, as with `updateInheritedAttributes`, the iteration should stop when we reach the value returned by `getEndIndex()`.

The corresponding functionality updating the `AGContext` objects while mapping is implemented in `GEGrammar::buildAGTree` method.

5.4 Example: Knapsack

Version 0.26.01 comes with one example implemented. This is implemented as in papers by Karim and Ryan. The example directory
`EXAMPLES/AttributeGrammar/Knapsack/GE_MITGALIB`
contains following files source files:

- `main.cpp`;
- `symbolclasses.h` and `.cpp`
- `AGLookUp.cpp`
- `knapsack.h`
- `knap-slang.cpp` and
- `GEknapsack.sl`
- `grammar.bnf`

Note, `GEListGenome.h/cpp` and `initfunc.cpp`are not specific to Attribute Grammars.

`main.cpp` contains the `main` function and is therefore responsible for executing the application. It also initialises the `mapper` object as in any libGE example. Also, it initialises `AGLookUp` and `AGContext` objects. The reason these objects are initialised in `main.cpp` is to allow the user to override the default implementation (in the case of `AGContext`) and/or give the mandatory problem specific implementation (as in the case of `AGLookUp`. Thus, the user can extend the functionality without having to recompile libGE.

`symbolclasses.h/.cpp` files implement the problem specific classes for the symbols specified in `grammar.bnf` file. The example provides separate classes for symbols `<S>`, `<K>`, and `<I>`. All the *item* symbols (`item(0); - item(e)`, are dealt with a single class called `Item` because these symbols only differ in the exact *values of their attributes* and not in terms of the *set of the attributes* that they required.

`AGLookUp.cpp` implements the problem specific look up functionality as required for the class `AGLookUp`.

`knap-slang.cpp` provides an interface with s-lang interpreter that evaluates the phenotypes for this problem.

`GEknapsack.sl` is an s-lang script file that evaluates the fitness of each *valid* phenotype generated by libGE. Note, that any invalid phenotype is detected before invoking the fitness function and is not passed on to this script.

`grammar.bnf` contains the grammar for this problem.