LEESA: Language for Embedded Query and Traversal

Sumant Tambe (sutambe [AT] dre [d0t] vanderbilt [d0t] edu)

Institute for Software Integrated Systems (ISIS), Vanderbilt University, Nashville, TN

(last updated Oct. 15, 2008)

Introduction

LEESA is about a radically new way of writing C++ UDM (Universal Data Model) based GME interpreters. LEESA (**L**anguage for **E**mbedded qu**E**ry and traver**SA**l) is a domain-specific embedded language (DSEL) in C++ that provides a succinct and expressive notation for writing object structure traversal. It decouples traversal from visitation actions and improves visitor reusability. By virtue of being declarative (partially), LEESA significantly reduces the development cost of programs operating on complex object structures (*e.g.*, domain-specific modeling language (DSML) interpreters) compared to the traditional techniques. LEESA is embedded in C++ using sophisticated generic programming techniques (Expression Templates)

Motivation and research documents

- A research paper on LEESA titled, "An Embedded Declarative Language for Hierarchical Object Structure Traversal" has been published in the 2nd International Workshop on Domain-Specific Program Development (DSPD), GPCE 2008, Nashville, Tennessee, October 22, 2008.
- Presentation slides for this paper are available here.

Purpose of this document is to provide a through documentation of the features and capabilities of LEESA with examples.

Downloading LEESA

The best way to obtain LEESA (and any updates) is from the CoSMIC repository using subversion. Subversion URL for downloading LEESA is below.

svn://svn.dre.vanderbilt.edu/CoSMIC/trunk/CoSMIC/Utils/LEESA

Software needed to use LEESA

LEESA is a *header-only* C++ library except in one case where it depends on Boost regular expression library. That means, including LEESA.h in your main program should suffice in most cases – no linking is needed. However, LEESA depends on following binary libraries that must be linked.

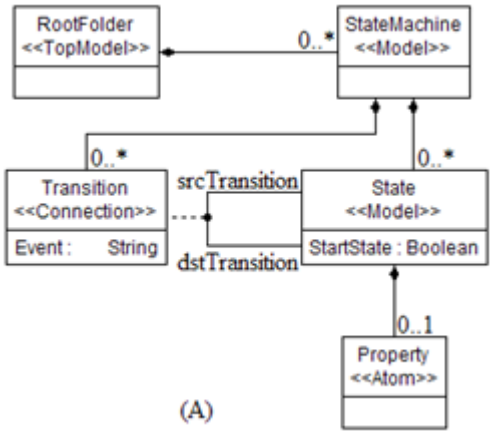
1. UDM (Universal Data Model) revision 2745 or later. LEESA does not support UDM 3.1.2 or any earlier versions of UDM. A binary version of UDM-r2745 can be downloaded from here. Sources of UDM can be downloaded using subversion using the following URL:

svn://svn.isis.vanderbilt.edu/MoBIES/UDM/trunk

2. Boost C++ Library 1.35 or later. Boost binaries can be downloaded from BoostPro consulting website. LEESA uses header-only libraries from Boost, however, some uses (described below) of LEESA may require linking with boost-regex library.

LEESA Language Documentation

The LEESA examples in this document are based on the StateMachine meta-model shown here. Some motivating examples based on this meta-model are provided in this presentation. It is highly recommended to go through these presentation slides *before* reading this document. This document is meant to be a reference for LEESA and assumes introductory level of familiarity with it.



Using LEESA in your UDM project

Using LEESA needs three steps.

1. Generate UDM classes from your *paradigm.xml* file using udm.exe with **–v** as one of the options. This will generate the base visitor class in the namespace of your paradigm name and accept member functions in every kind name when –v command line option is passed to it.

2. Before #including LEESA.h in your cpp file, #define a macro called **"PARADIGM_NAMESPACE_FOR_LEESA"** to the name of your paradigm. #Include LEESA.h **after** defining the macro. For example, consider that the StateMachine meta-model above is defined under paradigm HFSM. In that case, define:

```
#define   PARADIGM_NAMESPACE_FOR_LEESA    HFSM
#include  "LEESA.h"
```

   If PARADIGM_NAMESPACE_FOR_LEESA is not defined, but you still include LEESA.h, LEESA detects that and flags an error. Note that LEESA.h must be in your include path.

3. LEESA namespace should be opened by writing: `using namespace LEESA;` Without opening LEESA namespace, there is no way to leverage the syntactic sugar provided by it. And it is extremely hard to appreciate LEESA without its syntactic sugar.

   However, it is **strongly** advisable that LEESA namespace should be opened (every time) inside smaller scopes such as a function or a class rather than in the global namespace. If proper care is not exercised with LEESA namespace, C++ compiler may get confused with normal C++

statements and LEESA statements. This happens due to generic nature of templates in LEESA and a whole slew of overloaded operators that are also overloaded in some other places in C++. E.g., `std::cin` and `std::cout`. You can still write `cin, cout, sstream` statements while using LEESA as its implementation guards users against ambiguous overloaded operator errors in most cases but it can't do so for the operators that it does not know about!

Basic syntax and traversal strategies

1. Every LEESA expression begins with a kind name followed by empty brackets. The kind names are nothing but the classes generated by UDM from the meta-model. The first kind name need not be the `RootFolder` in every LEESA expression; it can be any other kind name except `MgaObject`, which is not supported by LEESA. E.g.,

   ```
   StateMachine() >> State() >> Property()
   ```

2. **Traversal strategies:** Kind names are separated by traversal strategies in every LEESA expression. LEESA defines two main traversal strategies: *breadth-first* and *depth-first*. Breadth-first strategy is specified using ">>" whereas depth-first strategy is specified using ">>=".

   a. Both the traversal strategies navigate direct parent child relationship. The kind name at the right hand side of the operator should be composed (immediately) inside the left hand side kind name. For instance, in the example above, `StateMachine` can occur at the left hand side of `State` but not vice versa. Moreover, `Property` cannot occur after `StateMachine` because `Property` is not a direct child of `StateMachine`. In other words, every parent/child (composition) relationship should be expressed explicitly in LEESA.

   b. **Breadth-first strategy:** Result of a valid LEESA expression written using breadth-first strategy *alone* is a collection (STL vector) of the last kind name in the expression. For example, the above expression results a `std::vector<Property>`. Breadth-first strategy progressively collects all the instances of a given child kind name as it navigates the parent/child relationship deeper.

   c. **Depth-first strategy:** The result of a valid LEESA expression written using one or more occurrences of the depth-first strategy operator (>>=) is –– please note –- **void.** for how Depth-first strategy, as the name suggests, navigates one instance at a time and goes deeper before moving on to the next instance of the same kind. It is not clear at this point what could be a meaningful return type for depth-first strategy because the order of elements it visits can't be grouped together easily (based on kind names) like in the breadth-first strategy. However, this strategy is most useful while using visitors.

   d. **Combining strategies:** Breadth-first and depth-first strategies can be combined together in a single LEESA expression. Result of such an expression is, again, **void.** It is observed that using multiple strategies in a single large LEESA expression hampers comprehensibility of the traversal.

3. **Traversing from child to parent:** LEESA uses "<<" operator to traverse composition relationship in reverse direction, i.e., from child to parent. For instance, following example yields parent state of a property. Please see the `Unique` query operator to see how it is useful while navigating from child to parent.

```
Property() << State()
```

4. **Traversing Associations:** LEESA is designed for object network traversal and therefore supports arbitrary user-defined association traversal. LEESA uses ">>&" operator to specify association traversal. For instance, in the `StateMachine` meta-model above, `Transition` is a user-defined association between two states. With respect to a `Transition` between two states, one state is a "source" state whereas the other state is a "destination" state. These source and destination roles are captured in the `Transition` class using `srcTransition` and `dstTransition` member functions. LEESA uses these functions as a way to navigate association. For instance, the example below returns a set of `States` those are at the "destination" end of all the `Transitions` under a `StateMachine`. The result of such an expression is the same as the return type of the association function being used.

```
StateMachine() >> Transition >>& Transition::dstTransition
```

Note that there is no empty bracket at the end of the association name.

Executing LEESA expressions

1. Every LEESA expression must be *evaluated* to obtain its result (which could be void in case of depth-first strategy.) To evaluate, LEESA provides a function by the same name. For example, following is the most convenient way of using LEESA in a C++ program.

```
RootFolder rf;
std::vector<State> sv =
LEESA::evaluate(rf, RootFolder() >> StateMachine() >> State());
```

The evaluate function takes two parameters. The first parameter should be a valid (non-null) UDM object whereas the second parameter should be a valid LEESA expression. The kind name of the first parameter must be the same as that of the first kind name in the LEESA expression. For example, `rf`, which is the first parameter is of `RootFolder` kind which is the first kind name in the expression.

Alternatively, the first parameter could be an object, a `std::set`, or a `std::vector` of the same kind name.

Labeling LEESA expressions

1. LEESA statements can be labeled before they are evaluated. LEESA uses `BOOST_AUTO` macro to label LEESA expressions. The `BOOST_AUTO` macro emulates the proposed `auto` keyword in C++. For example,

```
State state;
BOOST_AUTO(expr, State() >> Property());
std::vector<Property> vp = LEESA::evaluate (state, expr);
```

Using labeled LEESA expressions, compound expressions can be built. While building compound expressions, all the rules of composition stated earlier should be followed.

```
RootFolder rf;
BOOST_AUTO(get_statemachines, RootFolder() >> StateMachine());
BOOST_AUTO(get_states, get_statemachines >> State());
Std::vector<State> s = LEESA::evaluate (rf, get_states);
```

Query operators in LEESA

LEESA defines several query operators that can be used to perform various *actions* on the intermediate results of a LEESA expression. The first parameter of every query operator is a kind name that indicates the kinds that are being processed by the operator. This kind name must match the result of the previous expression. The query operators can also participate in expressions that use depth-first traversal strategy. Query operators do their job internally but the result of the expression is still void. All the query operators should follow after a ">>" operator.

1. **SelectByName (Kind, String)**

   Filters the instances whose name do not match the regular expression given as the second parameter. The second parameter string coule be a `const char *` or a `std::string` that represents a regular expression. Using this query operator requires linking with boost-regex library. Example:

   ```
   State() >> SelectByName(State(), "abcd") >> Property()
   ```

2. **SelectSubSet (Kind, vector<Kind> v)**

   Filters the elements that are not contained in vector v. The result of this query operator is a subset or the same set as v.

3. **Select (Kind, predicate p)**

   Filters the elements that do not satisfy predicate p. Predicate could be a standard C++ unary function or a unary function object. The types of the parameters of the function must match the kind.

4. **CastFromTo (From Kind, To Kind)**

   This query operator is a wrapper around `Udm::IsDerivedFrom` function. It selects elements only if `Udm::IsDerivedFrom` returns true.

5. **Sort (Kind, Comparator c)**

   As the name suggest, it sorts the result of the previous expression using `comparator c.` The types of parameters of the comparator function must match the kind.

6. **Unique (Kind, BinPred c)**

   As the name suggests, it selects only unique elements from the result of the previous expression. It uses a binary predicate c to determine uniqueness. A binary predicate could be a standard C++ function with two parameters or a binary function object. The parameters of the function must match the kind.

7. **Unique (Kind)**

   This operator uses the default uniqueness property defined by UDM to filter elements. This query operator is particularly useful while using "<<" operator that navigates composition relationship from child to parent. While navigating from child to parent, identical parent objects can be easily eliminated using this query operator. Example,

```
std::vector <State> states; // populate this vector somehow.
std::vector<StateMachine> sm =
evaluate (states, State() << StateMachine() >> Unique(StateMachine());
```

8. **ForEach (Kind, callback)**

   ForEach is a query operator that simply invokes a function on every element of the intermediate result of a LEESA expression. The `callback` is a function or a function object that accepts a single parameter of the same kind as `Kind` and returns void.

Results of all the query operators that are dependent on a predicate or comparison can be inverted using logical negation operator "!". It is a unary operator that occurs before the query operator name. For example, the following query will select states that do not have name "abcd".

```
State() >> ! SelectByName (State(), "abcd")
```

Using Visitors

One of the most important features of LEESA is its support for visitors. If a visitor is to visit kind K, then visitor object should be added after "K() >>". Any number of consecutive visitations are supported by LEESA. For instance, visiting all the states 3 times is possible using

```
CountVisitor cv;
StateMachine() >> State() >> cv >> cv >> cv;
```

Visiting multiple different kinds is possible in a similar fashion. Simply append ">>" and a visitor object after the desired kind name. Visitors can be used irrespective of the traversal strategy. Visitors make sense when combined with the depth-first strategy because, depth-first strategy guarantees that all the children will be traversed before moving on to the next element of the same kind. For example, in the traversal below, all the states and their properties will be visited before going to the next statemachine.

```
CountVisitor cv;
RootFolder() >>= StateMachine() >> cv >> State() >> cv >> Property() >> cv ;
```

An improved syntax for visitors is on its way in LEESA. Instead of using `>> visitor`, the new syntax will support `Kind()[visitor]`. Visitor object is written in a square bracket after kind name. LEESA already has the necessary infrastructure to support this syntax. However, to make this syntax work, some additional help from UDM is needed.

Visiting siblings

So far we talked about only parent-child and association traversal. An important aspect of traversal is visiting siblings. Siblings are kind names that are children of the same parent kind. LEESA supports sibling traversal using `MembersOf` construct. Return type of `MembersOf` construct is void. `MembersOf` construct should always be used with labeling support provided by BOOST_AUTO. For example, visiting Transition and State in every StateMachine in that order is done using:

```
CountVisitor cv;
BOOST_AUTO(v_tran, Transition() >> cv);
BOOST_AUTO(v_state, State() >> cv);
BOOST_AUTO(members, MembersOf(StateMachine(), v_tran  FOLLOWED_BY  v_state)
StateMachine() >>= members;
```

The keyword "`FOLLOWED_BY`" need not be used literally but reads well! If the expressions to the left or right hand side of `FOLLOWED_BY` keyword are complex statements, (those which use any of breadth-first, depth-first, visitor, or association traversal) then use of `BOOST_AUTO` for those expressions is highly recommended and must in some cases.

Coding Guidelines

1. "`using namespace LEESA;`" should be put inside the smallest possible scope and should be repeated in other scopes, if needed, to prevent polluting global namespace.
2. `MembersOf` statement should always be labeled using `BOOST_AUTO`.
3. Avoid inline complex statements around `FOLLOWED_BY`. Use statement labeling using `BOOST_AUTO` instead.
4. Build your queries and traversals incrementally while making small changes each time. Make sure your program compiles after every small change you make to the expression.

Inside LEESA

The core of LEESA is Expression Templates and C++ operator overloading on steroids! Understanding C++ operator precedence and associativity may help you understand and (hopefully!) appreciate LEESA's design decisions. A partial table (relevant to LEESA) of operator precedence and associativity in C++ is shown below. Based on these rules, LEESA tries to seek a fine balance between expressiveness and what is possible in C++.

Precedence	Operator	Description	Associativity
1 (higher)	::	Scoping operator	none
2	[ ] ()	Array access, grouping operator	Left to right
3	& !	Address of, logical negation	Right to left
7	<< >>	Bitwise shift left, bitwise shift right	Left to right
13	&&	Logical AND	Left to right
16 (lower)	>>=	Bitwise shift right and assign	Right to left

Generally, it is not advisable to write long, complex LEESA expressions that mix the operators below freely. It makes traversals extremely counterintuitive to understand due to complex rules of how and in which order compiler resolves the precedence and associativity of these overloaded operators. Therefore, expression labeling using BOOST_AUTO and LEESA's readability enhancing macros shown below are your friends!

```
#define MembersOf(A,B) A && B
#define DEPTH_FIRST    >>=
#define BREADTH_FIRST  >>
#define FOLLOWED_BY    &&
```