An Embedded Declarative Language for Hierarchical Object Structure Traversal

Sumant Tambe and Aniruddha Gokhale
Vanderbilt University, Nashville, TN, USA
{sutambe,gokhale}@dre.vanderbilt.edu

Abstract

A common challenge in processing large domain-specific models and in-memory object structures (*e.g.,* complex XML documents) is writing traversals and queries on them. Object-oriented (OO) designs, particularly those based on the Visitor pattern, are commonly used for developing traversals. However, such OO designs limit the reusability and independent evolution of visitation actions (*i.e.,* the actions to be performed at each traversed node) due to tight coupling between the traversal logic and visitation actions, particularly when a variety of different traversals are needed. Code generators developed for traversal specification languages alleviate some of these problems but their high cost of development is often prohibitive. This paper presents **L**anguage for **E**mbedded qu**E**ry and traver**SA**l (LEESA), which provides a generative programming approach for embedding object structure queries and traversal specifications within a host language, C++. By virtue of being declarative, LEESA significantly reduces the development cost of programs operating on complex object structures compared to the traditional techniques.

1. Introduction

A common practice for providing access to a large hierarchical data consisting of heterogeneous types of objects is to provide type-safe interfaces for every different type of object. These object structures are represented in memory as a hierarchical collection of typed objects with accessors and mutators. A canonical example of this approach is XML data binding (*e.g.,* Java Architecture for XML Binding (JAXB)), where a code generator generates a hierarchy of classes representing types in an XML schema in such a way that de-marshaling of an XML document conforming to the schema reduces to simply instantiating the root class of the generated class hierarchy.

An identical approach [1] is adopted in the development of domain-specific modeling (DSM) [2] tools such as analysis tools, code generators, model checkers, model transformation tools, among others. In the DSM methodology, domain-specific models conforming to a meta-model are created. Often the interpretation of these domain-specific models require navigation and querying capabilities over the complex object structures governed by the underlying meta-model.

Extracting meaningful data from such object structures often requires different traversals over the object structure. The Visitor [3] pattern is a preferred approach for writing such traversals since it groups together related visitation actions (*i.e.,* the actions to be performed at each traversed node) in the visitor class whereas the traversals are localized in the concrete classes of the object structure. The consequence of using the Visitor pattern is a rigid traversal of the object structure, which is inefficient when the desired traversal requires visitation to only a subset of elements.

Alternatively, the traversal algorithm could be put in the visitor, which results into not only duplicating the traversal code in each concrete visitor but also coupling it with the visitation actions. Introducing Iterators [3] can eliminate the dependence on the underlying physical data structure (*e.g.,* linked-list, vector) used by the containers, however, dependence on the order of types visited still remains in the visitor classes. Such coupling of traversal and visitation actions adversely affects the reusability of the visitors, especially when visitation actions are stable but traversals are not.

Domain-specific languages (DSL) [4–7] that are tailored for the traversals of object structures have been proposed to overcome the pitfalls of the Visitor pattern. These DSLs separate traversals from the visitation actions. In general, the basic building block of these languages are one or more traversal rules where every rule consists of a composite type and its children types listed in the desired traversal order. A code generator transforms the traversal rules into a conventional procedural language program. For example, the Traversal/Visitor Language (TVL) [6] can be used to write traversal rules for domain-specific models. From the traversal rules, TVL's language processor generates visitor-based C++ code that visits the object structure in the specified order. Finally, a separate step is needed to compile the generated code with the manually written visitation actions.

In spite of significant research in developing traversal DSLs, the existing approaches have not enjoyed wide spread use. Among the most important reasons [8–10] hindering their adoption are (1) high upfront cost of the language and tool development, and (2) their evolution and maintenance overhead. Development of language tools such as a code generator requires the development of at least a lexical analyzer, parser, back-end code synthesizer and a pretty printer. Moreover, Mernik et al. [8] claim that language extension is hard to realize because most language processors are not designed with extension in mind. This increases the maintenance overhead of the code generator as the DSL and the back-end programmatic interface evolve over time.

Apart from traversal, querying is another commonly performed operation on hierarchical object structures, which, although possible, is not supported in the existing [5, 6] DSLs for traversal specification. Querying requires support for selection, filtering, sorting, and folding based on user-defined predicates, which significantly increases the implementation complexity and the learning curve of

the DSLs. Lack of integrated support for querying and traversal, however, reflects in the increased implementation complexity of the visitors because the visitors must incorporate queries using procedural techniques. Moreover, the user-defined predicates used in the queries are often tightly coupled with the object structure.

To address the challenges outlined above, in this paper we present a generative programming [11] approach to develop a domain-specific embedded language (DSEL) [12] for querying and implementing multiple visitor-based traversals over domain-specific models. Our approach is not limited to domain-specific models only; and it is applicable to a general class of problems for hierarchical object structure traversal. We have developed an embedded DSEL for C++ called **L**anguage for **E**mbedded qu**E**ry and trave**r**S**A**l (LEESA), which leverages C++ templates and operator overloading to provide an intuitive syntax for writing queries and traversals. LEESA makes the following contributions:

- It embodies a novel, low-cost approach of developing a DSEL suitable for object structure query and traversal in a multi-paradigm language such as C++. The generative programming approach adopted by LEESA's language processor does not require development of a complex lexer, parser and code generator because it leverages the C++ compiler to accomplish its goals and thereby significantly reduces [8–10] the cost of DSEL development.
- It enables strong separation between the code that performs traversal and the actions to be performed at each traversed node. This prevents the knowledge of the object structure from being tangled throughout the code.
- The notation provided by LEESA is declarative, which focuses on *which* objects are visited but hides the details of *how* they are visited. Moreover, the notation is suitable for writing queries and visitor-based traversals simultaneously over an object structure, unlike any previous approach. The result is a succinct yet expressive notation for traversal.

The remainder of the paper is organized as follows. Section 2 presents the architecture of LEESA and describes how we resolve the challenges in designing a notation for object structure querying and traversal; Section 3 describes in detail our generative programming approach of embedding LEESA's language processor in C++; Section 4 presents open issues that require further investigation; Section 5 presents related work; and Section 6 concludes the paper.

2. LEESA: Language for Embedded QuEry and TraverSAl

We begin this section with the layered architecture of LEESA and describe how LEESA's architecture supports the solutions for (1) embedding a notation for object structure traversal and (2) implementing a language processor in the host language, C++. We further describe the challenges faced while developing LEESA in the context of a small case study of a finite state machine (FSM) modeling language.

2.1 Layered Architecture of LEESA

Figure 1 shows the four layer architecture of LEESA. At the bottom is the hierarchical *object structure* we would like to search and traverse. For example, it could be an instance of a domain-specific model or a large XML document. A *generic data access layer* is a layer of abstraction over the object structure, which provides an interface for accessing the elements in the object structure, iteratively. Often, a meta-level code generator is used to generate the classes that provide such an interface. Several different types of meta-level code generators such as XML schema compilers (*e.g.,* JAXB) and domain-specific modeling tool-suites [1] exist that generate a set of classes from the meta information of the object structures.

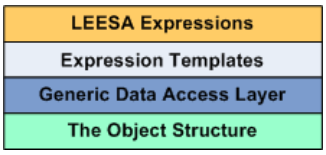

Figure 1. Layered Architecture of LEESA

Expression Templates [13, 14], is the key idea behind embedding a declarative notation for query and traversal in C++. Expression templates map the *LEESA expressions*, embedded in a C++ program onto the generic data access layer. Expression templates raise the level of abstraction by hiding away the iterative process of accessing objects and focus only on the types of the objects and different strategies of traversal. Moreover, LEESA's expression templates are generic, which can be used for writing queries and traversals over any hierarchical object structure provided an appropriate generic data access layer is provided.

2.2 Finite State Machine Language: A Case-study

We use a FSM modeling language as a running example throughout the paper to describe various capabilities of LEESA. Figure 2 shows a meta-model of a FSM language using a UML-like notation. Our FSM meta-model consists of a `StateMachine` with one or more `States` having directional edges between them called `Transitions`. One of the states can be marked as a *start state* using a boolean attribute. States may optionally contain a `Property` element representing an arbitrary domain-specific value (*e.g.,* timeout period in seconds). A `Transition` represents an association between two states, where the source state is in the `srcTransition` role and the destination state is in the `dstTransition` role with respect to a `Transition` as shown in Figure 2. The `RootFolder` is a singleton that represents the root level model.

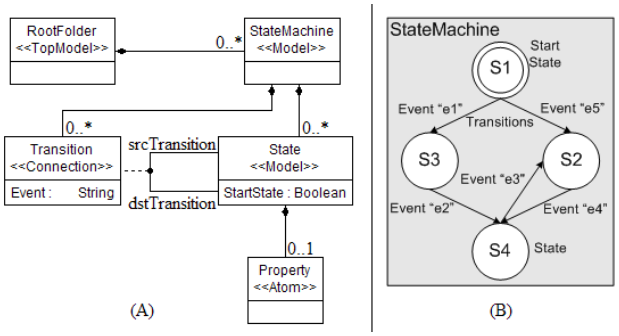

Figure 2. (A) Meta-model of Finite State Machine (FSM) Language (B) A Simple FSM Model

We used Universal Data Model (UDM) [1] – a code generator for developing domain-specific modeling tools – to generate the data access layer for iteratively accessing FSM models. The generic data access layer for FSM models consists of five C++ classes: `RootFolder`, `StateMachine`, `State`, `Transition`, and `Property`.

2.3 Notation for Object Structure Querying

Designing an intuitive domain-specific notation for a DSL is central to achieving productivity improvements as domain-specific nota-

tions are closer to the problem domain in question than the notation offered by general-purpose programming languages. The notation should be able to express the key abstractions and operations in the domain succinctly so that the DSEL programs become more readable and maintainable than the programs written in general-purpose programming languages. For object structure querying, the key abstractions are the objects and typed collections of the objects while the basic operation performed is navigation of associations (mostly composition) from one type of object to another.

An important constraint imposed by the host language while designing a notation for a DSEL is to remain within the limits of the user-definable operator syntax offered by the host language. Quite often trade-offs must be made to seek a balance between the expressiveness and intuitiveness of the embedded notation against the ease of implementation. In the following, we describe the notation we chose for LEESA that can be embedded in a C++ program without sacrificing its expressiveness.

Resolution. Listing 1 shows an example of a LEESA query that returns a set of all the `Properties` in the model rooted at the `RootFolder`. This notation is an immediate improvement over traditional iterative techniques of querying objects. Using the traditional approach, a programmer will have to write at least two nested *for* loops: one to iterate over all the `StateMachines` under the `RootFolder` and another to iterate over all the `States` under each `StateMachine`. Finally, a set of `Properties` must also be populated iteratively.

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

Listing 1: A LEESA Expression to Find All The Properties in a FSM Model

The example in Listing 1 is based on the composition relationship only as every type (except `RootFolder`) is strictly contained inside its left side type. However, LEESA is capable of traversing other arbitrary associations defined between different types of objects. For example, Listing 2 shows a LEESA query that returns all the states that have at least one incoming transition. Note that such a set can be conceptually visualized as a set of states that are at the *destination* end of a transition.

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

Listing 2: A LEESA Expression to Find All The `States` With an Incoming `Transition`

To improve the readability of LEESA expressions, several additional syntactic elements are introduced in Listing 2, which are designed to easily distinguish between the composition associations and user-defined associations in a LEESA expression. The operator "$\gg$&" is used to find objects at either ends of an association provided the expression to the left of the operator yields a set of associations. For example, the expression to the left side of the operator $\gg$& in Listing 2 yields a set of `Transitions`, which represents associations between `States`. The remaining expression to the right side of the $\gg$& operator, returns a `State` that is in the `dstTransition` role with respect to every `Transition` in the previously obtained set. Finally, LEESA provides support for navigating composition relationships in the reverse (*i.e.,* from child to parent) direction using the left shift (*i.e.,* "$\ll$") notation.

2.4 Supporting Intermediate Result Processing Using Query Operators

Writing queries over object structures often requires processing the intermediate results before the rest of the query is executed (*e.g.,* filtering objects that do not satisfy a user-defined predicate, sorting objects using user-defined comparison functions.)

Resolution. LEESA provides a set of query operators that process the intermediate results produced by the partial execution of the query. These query operators are in fact higher-order functions that take user-defined predicates or comparison functions as parameters and apply them on a collection of objects.

```
int comparator (State, State) { ... }
bool predicate (Property) { ... }
RootFolder() >>
StateMachine() >> SelectByName(StateMachine(),"C.*")
>> State()    >> Sort(State(), comparator)
>> Property() >> Select(Property(), predicate)
```

Listing 3: A LEESA Expression Written Using Query Operators

Listing 3 shows an example of a LEESA query that uses three predefined query operators: `SelectByName`, `Sort`, and `Select`. The `SelectByName` operator accepts a regular expression as its second parameter and returns a collection of objects of type `StateMachine` that have names that match the regular expression. The `Sort` function, as the name suggests, sorts a collection using a user-defined comparator. Finally, `Select` filters out objects that do not satisfy the user-defined predicate. LEESA can be extended easily to support additional operators if needed.

The result of the query in Listing 3 is a set of `Properties`, however, the intermediate results are processed by the query operators before navigating composition relationships further. Therefore, the final or any intermediate result of the query could be a null set. `Sort` and `Select` are examples of higher-order functions that accept conventional functions as parameters as well as stateful objects that behave like functions, commonly known as *functors*.

2.5 Adding Visitors: Combining Querying with Traversal

Although the object structure querying capability is useful for obtaining a collection of objects, implementing the notion of *traversal* with querying is hard because traversal involves not only selecting the right set of objects, but also performing a set of operations on those objects in a specific order. As mentioned before, we call such actions to be performed at each traversed object as visitation actions. It is desirable to modularize visitation actions without overly coupling them with the order of traversal. Moreover, it should be possible to reuse and extend the visitation actions without having to deal with the object structure or the order of traversal.

Resolution. The Visitor [3] pattern has been successfully used to create extensible designs where no changes to the object structure are necessary to add new visitation actions. Therefore, we combine the declarative querying capability of LEESA with first class support for the Visitor pattern. Listing 4 shows a LEESA expression that combines a visitor of type `PrintVisitor` with the query shown in Listing 3.

```
PrintVisitor pv; // instantiate a visitor
RootFolder() >>
StateMachine()>> SelectByName(StateMachine(),"C.*")>>pv
>> State()    >> Sort(State(), comparator)    >> pv
>> Property() >> Select(Property(), predicate)
```

Listing 4: Combining A LEESA Expression With a Visitor

The `PrintVisitor` has been implemented as a concrete C++ class that inherits from an abstract class `Visitor` as shown in Listing 5. When instances of subtypes of `Visitor` are combined with LEESA expressions, appropriate `Visit*` functions of the visitor

are invoked where specified. This also allows visitor objects to accumulate state during traversal. For example, in the LEESA expression shown in Listing 4, `Visit_StateMachine` and `Visit_State` member functions of `pv` (an object of type `PrintVisitor`) are invoked on the collections of objects returned by the query operators `SelectByName` and `Sort`, respectively.

```
class PrintVisitor : public Visitor {
  public:
    /// ... constructors, members, if any.
    virtual void Visit_StateMachine(StateMachine);
    virtual void Visit_State(State);
    virtual void Visit_Property(Property);
    virtual void Visit_Transition(Transition);
};
```

Listing 5: Definition of the PrintVisitor Class in C++

As described in Section 1, the coupling between visitors and the object structure increases if different visitors require different traversals. However, our approach of combining LEESA with a visitor is an improvement over the traditional Visitor pattern because the visitor implementation is completely modularized away from the object structure and the traversal order. The `PrintVisitor` class in Listing 5 deals purely with its associated visitation actions. Any change in the traversal order or the query operators does not affect the `PrintVisitor` class in any way. This loose coupling between the traversal and visitation actions facilitates reuse of the visitor implementation in future even when the new visitor is based on a different order of traversal.

2.6 Visiting Sibling Types

Composition of multiple types of objects in a composite object is commonly observed in practice. For example, the FSM language has a composite called `StateMachine` that consists of two types of siblings: `State` and `Transition`. Support for object structure traversal in LEESA would not be complete unless support for visiting multiple types of siblings is provided. Moreover, such a decision of visiting siblings could be made dynamically depending upon the result of some predicate. Therefore, it is desirable to support selection of the course of traversal based on conditionals.

Resolution. Listing 6 shows an example of how LEESA supports sibling traversal. The example query visits all the `States` in a `StateMachine` before all the `Transitions`. The `MembersOf` notation is designed to improve readability as its first parameter is the common parent (*i.e.,* `StateMachine`) followed by a comma separated list of LEESA subexpressions for visiting the members of the parent in the given order. LEESA also supports `Branch` notation that accepts a predicate as a parameter and depending upon the result of the predicate, one of the two possible traversal are chosen. However, we do not discuss the `Branch` notation further due to space considerations.

```
PrintVisitor pv;
RootFolder() >> StateMachine()
  >> MembersOf(StateMachine(), State()      >> pv,
                               Transition() >> pv)
```

Listing 6: A LEESA Expression for Traversing Siblings: `States` and `Transitions`

2.7 Supporting Flexible Strategies of Traversal

The examples of traversals presented in earlier sections and Listing 7 are designed to perform a breadth-first traversal of the object structure. For example, using the LEESA expression in Listing 7, all the `StateMachine` objects are visited before all the `State` objects and all the `State` objects are visited before all the `Property` objects. However, for certain applications such as, serializing a FSM model into XML form, requires depth-first traversal of the structure. Therefore, it is desirable to have a notation that supports multiple strategies for traversal and combines them in ways that are suitable for the problem at hand.

```
PrintVisitor pv;
RootFolder() >> StateMachine()   >> pv
             >> State()          >> pv
             >> Property()       >> pv
```

Listing 7: A LEESA Expression for Breadth-first Traversal of an FSM Model

Resolution. Depth-first traversal is expressed using operator "$\gg=$" in LEESA. For example, Listing 8 shows a LEESA expression that traverses all the objects in depth-first order. In particular, when a `StateMachine` object is visited by the `PrintVisitor`, all of its children `States` and grandchildren `Property` objects are visited before moving on to the next `StateMachine` object. Similarly, `Property` object, if any, of each `State` object is visited before visiting the next `State` object. Finally, it is also possible to combine breadth-first traversal notation with depth-first traversal notation in a LEESA expression.

```
PrintVisitor pv;
RootFolder() >>= StateMachine()   >>= pv
             >>= State()          >>= pv
             >>= Property()       >>= pv
```

Listing 8: A LEESA Expression for Depth-first Traversal of an FSM Model

3. Embedding LEESA's Language Processor in C++

In this section we describe how LEESA expressions shown in the previous section are embedded in a C++ program using programming idioms such as operator overloading and Expression Templates [13, 14].

```
class RootFolder {
   template <class T> set<T> children ();  };
class StateMachine {
   template <class T> set<T> children ();  };
class State {
   template <class T> set<T> children ();  };
class Transition {
   State srcTransition();
   State dstTransition();
};
class Property;
```

Listing 9: C++ Classes Generated by The UDM Tools

3.1 Realization of Generic Data Access Layer

In our existing implementation of LEESA, we used the programmatic interface provided by the Universal Data Model (UDM) [1] framework, which defines a development process and provides a set of supporting tools that are used for development of domain-specific modeling tools. The UDM tools generate the C++ classes shown in Listing 9 for the FSM meta-model shown in Figure 2. Each of these classes define a member function template that returns a set of children objects. For example, when the `children()` member function in class `StateMachine` is parameterized with

State, it returns a set of States whereas it returns a set of Transitions when the function is parameterized with Transition.

```
RootFolder::children<StateMachine>()
StateMachine::children<Transition>()
Transition::dstTransition()   //returns State
```

Listing 10: Functions Invoked While Executing the LEESA Query in Listing 2

Using the generic interface shown in Listing 9, the LEESA expression shown in Listing 2, is mapped to a sequence of function calls shown in Listing 10. To bridge the gap between the declarative specifications in LEESA and the underlying generic data access layer, we use expression templates which are described next.

3.2 Expression Templates: Bridging LEESA and the Object Structure

Expression Templates [13, 14] is a powerful generic programming technique in C++ that allows lazy evaluation of expressions which is not supported natively in C++. Lazy evaluation is important for LEESA because LEESA expressions, which are declarative query specifications over an object structure, can be passed as parameters to functions in a C++ program to extract results when needed. The computation required to extract the results is embodied in a rather complex C++ type that behaves like a function at runtime, commonly known as *functors*. Construction of such complex C++ types is enabled using repetitive instantiations of a function template combined with the ability of C++ to redefine certain built-in operators for user-defined types, known as *operator overloading*.

```
template <class L, class H>
ChainExpr<L, GetChildren<L, H> > // return type
operator >> (L l, H h) {
  typedef GetChildren<L, H> Operator;
  return ChainExpr<L, Operator>(l, h);
}
```

Listing 11: An Example of an Expression Template in LEESA

Listing 11 shows an example of an overloaded $\gg$ operator function template in LEESA. The function is parameterized with two type parameters (L and H) and returns a composite type that is parameterized with both the formal parameter types. The key idea in expression templates is compile-time recursive objects [15], which are instances of class templates that contain other instances of the same template as member variables. A rather clever arrangement of the parameters and the return type of a function template is used in such a way that repetitive instantiations of the template create multiple compile-time recursive objects resulting into an abstract syntax tree (AST) of types.

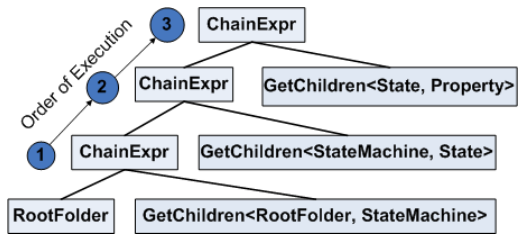

Figure 3. An Abstract Syntax Tree Created During Compilation of LEESA Expression Template in Listing 11

An example of an AST created while compiling the LEESA expression in Listing 1 is shown in Figure 3. GetChildren is one of the operators defined in LEESA that invokes the children() function on an object of its first parameter type (L) and yields a set of children of its second parameter type (H). Finally, the ChainExpr acts like a Composite [3], which allows construction of large LEESA expressions from smaller subexpressions.

The AST is evaluated in the order shown in Figure 3, which produces a set of Properties as its end result. In step (1), a set of StateMachines is obtained by applying GetChildren operator on the RootFolder. In step (2), the GetChildren operator is applied again on every StateMachine object yielding a set of States. Finally, in step (3), a set of Properties is obtained by applying the same operator on the set of States.

4. Open Issues

We discuss two issues: the first focuses on the relationship of LEESA with Adaptive Programming [4] while the second focuses on effective error reporting in LEESA.

Adaptive Programming presents an idea of "structure shyness" [4], which allows one to specify traversal strategies that ignore certain parts of the concrete structure and focus on the essential structure only. This lowers the coupling of code from the concrete structure and results in programs that are more resilient to the changes in the object structure. LEESA allows separation of traversal specifications from visitation actions, however, the traversal specifications are not "structure-shy" since all the intermediate concrete parts of the structure must be mentioned in a LEESA traversal specification. Naturally, it opens a question whether it is possible to ignore non-essential parts of the structure using an embedded language such as LEESA. Towards this end, we are investigating different approaches to substitute *wildcard* expressions in LEESA.

A second issue deals with effective error reporting of syntactic as well as semantic violations to improve usability of LEESA. DSELs are often unable to produce intuitive messages to the user indicating the exact location and the cause of the violations because they reuse the language processor of the host language, which has little or no knowledge of the syntax and semantics of the expressions in the embedded language. As a result, error reporting is often in terms of the host language artifacts instead of the DSEL artifacts. LEESA being an embedded language in C++ is no exception to this rule. However, we are investigating a novel approach to mitigate this inherent difficulty using new features in C++ that have been voted into the upcoming ISO standard for C++: C++0x.

C++0x has included *Concepts* [16] to express the syntactic and semantic behavior of types and to constrain the type parameters in a C++ template. Moreover, using concepts, the compiler can produce user-friendly error messages when types fail to satisfy template constraints. We are investigating how concepts might be used to express constraints on the LEESA expression templates to produce meaningful error messages when the constraints are not satisfied. Our initial results are promising.

5. Related Work

Significant research on domain-specific languages for object structure traversal exists. S-XML [17] and XML Translation Language [18] are functional languages embedded in Scheme and Haskell, respectively, for creating and processing XML-like trees. These DSELs are designed to exploit the functional characteristics of their host languages. Particularly, S-XML is a parenthesized language just like Scheme. LEESA, however, is based on C++, which is a procedural language. Moreover, LEESA has first class support for visitor-based designs with different traversal strategies, unlike the above-mentioned functional languages.

Gray et al. [6] and Ovlinger et al. [5] present an approach in which traversal specifications are written in a specialized language separate from the visitation actions. A code generator is used to

transform the traversal specifications into procedural code based on the Visitor pattern. This approach, as mentioned before, is heavy-weight compared to the embedded approach because it incurs a high cost for the development and maintenance of the language processor.

Language Integrated Query (LINQ) [19] is a Microsoft technology used to support SQL-like queries natively in a program to search, project and filter data in arrays, XML, relational databases, and other third-party data sources. LINQ expressions are designed to be embedded in .NET languages, particularly C#. We believe that LINQ is confined to query capabilities only and does not support visitor-based traversal like LEESA. Learning from the success of LINQ, the design goals of LEESA have included bringing a small subset of LINQ capabilities, such as, hierarchical object structure querying to standard C++, provided an appropriate data access layer is available.

Czarnecki et al. [9] compare staged interpreter techniques in MetaOCaml with the template-based techniques in Template Haskell and C++ to implement DSELs. Two approaches – type-driven and expression-driven – of implementing a DSEL in C++ are presented with an example from the domain of scientific computing. Within this context, our approach in LEESA is an expression-driven approach of implementing DSELs. Spirit(`http://spirit.sourceforge.net`) and Blitz++ [20] are two other prominent examples of expression-driven DSELs in C++. Spirit is an object-oriented recursive descent parser framework whereas Blitz++ is a high-performance scientific computing library with support for mathematical abstractions such as, dense vectors and multidimensional arrays. Although LEESA shares a significant number of implementation techniques with the above DSELs, querying and traversal of in-memory hierarchical object structure cannot be developed using Spirit or Blitz++.

Finally, Hofer et al. [21] propose an approach to provide multiple interpretations for a DSEL using *polymorphic embedding* whereas Seefried et al. [10] propose a way of optimizing DSELs using template techniques. We are investigating these approaches to enhance the applicability of LEESA to different hierarchical data sources as well as to improve its efficiency.

6. Conclusion

In this paper we reinforced the fact that domain-specific languages (DSLs) raise the level of abstraction closer to the domain in question using language constructs (notation) and their associated semantics, which results into higher productivity. We presented a case for *embedding* as a significantly cost-effective way of implementing a DSL particularly in the domain of hierarchical object structure traversal where mature implementations of iterative data access layer abound. Reduction in the development cost is mainly attributed to the reuse of the host language infrastructure such as a lexer, parser, code generator, and standard libraries, which must be developed from scratch in the non-embedded DSL approach.

To show the feasibility of our approach, we developed the **L**anguage for **E**mbedded qu**E**ry and trave**r****SA**l (LEESA), which is an embedded DSL in C++ for hierarchical object structure traversal. LEESA improves the modularity of the programs operating on complex hierarchical object structures (*e.g.,* domain-specific models) by separating the knowledge of the object structure from the actions performed when the nodes of interest are visited. LEESA provides equivalent modularization capabilities to that of the existing non-embedded DSLs while keeping the cost of DSL development as low as possible. We are investigating how usability of LEESA can be improved by proper domain-specific error reporting by combining new C++ features such as Concepts [16] with established generative programming techniques such as Expression Templates [13, 14].

LEESA is available in open source at `https://svn.dre.vanderbilt.edu/viewvc` under CoSMIC subversion repository.

References

[1] E. Magyari and A. Bakay and A. Lang and T. Paka and A. Vizhanyo and A. Agrawal and G. Karsai: UDM: An Infrastructure for Implementing Domain-Specific Modeling Languages. In: The 3rd OOPSLA Workshop on Domain-Specific Modeling. (October 2003)

[2] Gray, J., Tolvanen, J., Kelly, S., Gokhale, A., Neema, S., Sprinkle, J.: Domain-Specific Modeling. In: CRC Handbook on Dynamic System Modeling, (Paul Fishwick, ed.). CRC Press (May 2007) 7.1–7.20

[3] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA (1995)

[4] Lieberherr, K.J.: Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns. PWS Publishing Company (1996)

[5] Ovlinger, J., Wand, M.: A Language for Specifying Recursive Traversals of Object Structures. SIGPLAN Not. **34**(10) (1999) 70–81

[6] Gray, J., Karsai, G.: An Examination of DSLs for Concisely Representing Model Traversals and Transformations. In: 36 th Hawaiian International Conference on System Sciences (HICSS), Big Island, Hawaii, January 6-9, 2003. (2003)

[7] Crew, R.F.: ASTLOG: A language for examining abstract syntax trees. In: Proceedings of the USENIX Conference on Domain-Specific Languages, Oct. 1997. 229–242

[8] Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. (2005)

[9] Czarnecki, K., Striegnitz, J., Taha, W.: DSL implementation in MetaOCaml, Template Haskell, and C++. In: Domain Specific Program Genearation 2004. (2004)

[10] Seefried, S., Chakravarty, M., Keller, G.: Optimising Embedded DSLs using Template Haskell (2003)

[11] Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley, Reading, Massachusetts (2000)

[12] Hudak, P.: Building domain-specific embedded languages. ACM Computing Surveys (1996)

[13] Langer, A., Kreft, K.: C++ Expression Templates: An Introduction to the Principles of Expression Templates. Dr. Dobb's J. (March 2003)

[14] Veldhuizen, T.: Expression Templates. C++ Report (June 1995)

[15] Järvi, J.: Compile Time Recursive Objects in C++. In: TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems. (1998)

[16] Gregor, D., Järvi, J., Siek, J., Stroustrup, B., Reis, G.D., Lumsdaine, A.: Concepts: linguistic support for generic programming in C++. In: Proceedings of the Object Oriented Programming, Systems, Languages, and Applications (OOPSLA). Volume 41. (2006)

[17] Clements, J., Felleisen, M., Findler, R., Flatt, M., Krishnamurthi, S.: Fostering Little Languages. Dr. Dobb's J. (March 2004)

[18] Wallace, M., Runciman, C.: Haskell and XML: Generic combinators or type-based translation? In: Proc. of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99)

[19] Anders Hejlsberg et al.: Language Integrated Query (LINQ). http://msdn.microsoft.com/en-us/vbasic/aa904594.aspx

[20] Veldhuizen, T.L.: Arrays in blitz++. In: Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98). LNCS, Springer-Verlag (1998)

[21] Hofer, C., Ostermann, K., Rendel, T., Moors, A.: Polymorphic Embedding of DSLs. In: Proc. of Generative Programming and Component Engineering (GPCE). (2008)