LEESA: Embedding Strategic and XPath-Like Object Structure Traversals in C++

Sumant Tambe and Aniruddha Gokhale

Electrical Engineering and Computer Science Department,
Vanderbilt University, Nashville, TN, USA
{sutambe,gokhale}@dre.vanderbilt.edu

Abstract. Traversals of heterogeneous object structures are the most common operations in *schema-first* applications where the three key issues are (1) separation of traversal specifications from type-specific actions, (2) expressiveness and reusability of traversal specifications, and (3) supporting structure-shy traversal specifications that require minimal adaptation in the face of schema evolution. This paper presents **L**anguage for **E**mbedded qu**E**ry and traver**SA**l (LEESA), which provides a generative programming approach to address the above issues. LEESA is an object structure traversal language embedded in C++. Using C++ templates, LEESA combines the expressiveness of XPath's axes-oriented traversal notation with the genericity and programmability of Strategic Programming. LEESA uses the object structure meta-information to statically optimize the traversals and check their compatibility against the schema. Moreover, a key usability issue of *domain-specific error reporting* in embedded DSL languages has been addressed in LEESA through a novel application of *Concepts*, which is an upcoming C++ standard (C++0x) feature. We present a quantitative evaluation of LEESA illustrating how it can significantly reduce the development efforts of schema-first applications.

1 Introduction

Compound data processing is commonly required in applications, such as program transformation, XML document processing, model interpretation and transformation. The data to be processed is often represented in memory as a heterogeneously typed hierarchical object structure in the form of either a tree (*e.g.,* XML document) or a graph (*e.g.,* models). The necessary type information that governs such object structures is encoded in a schema. For example, XML schema [1] specifications are used to capture the vocabulary of an XML document. Similarly, metamodels [2] serve as schema for domain-specific models. We categorize such applications as *schema-first* applications because at the core of their development lie one or more schemas.

The most widespread technique in contemporary object-oriented languages to organize these schema-first applications is a combination of the Composite and Visitor [3] design patterns where the composites represent the object structure

W.M. Taha (Ed.): DSL 2009, LNCS 5658, pp. 100–124, 2009.

and visitors traverse it. Along with traversals, *iteration*, *selection*, *accumulation*, *sorting*, and *transformation* are other common operations performed on these object structures. In this paper, we deal with the most general form of object structures, *i.e.,* object graphs, unless stated otherwise.

Unfortunately, in many programming paradigms, object structure traversals are often implemented in a way such that the traversal logic and type-specific computations get entangled. Tangling in the functional programming paradigm has been identified in [4]. In object-oriented programming, when different traversals are needed for different visitors, the responsibility of traversal is imposed on the visitor class coupled with the type-specific computations. Such a tight coupling of traversal and type-specific computations adversely affects the reusability of the visitors and traversals equally.

To overcome the pitfalls of the Visitor pattern, domain-specific languages (DSL) that are specialized for the traversals of object structures have been proposed [5,6]. These DSLs separate traversals from the type-specific computations using *external* representations of traversal rules and use a separate code generator to transform these rules into a conventional imperative language program. This two step process of obtaining executable traversals from external traversal specifications, however, has not enjoyed widespread use. Among the most important reasons [7, 8, 9, 10] hindering its adoption are (1) high upfront cost of the language and tool development, (2) their extension and maintenance overhead, and (3) the difficulty in integrating them with existing code-bases. For example, 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. [7] claim that language extension is hard to realize because most language processors are not designed with extension in mind. Finally, smooth integration with existing code-bases requires an ability of not only choosing a subset of available features but also incremental addition of those features in the existing code-base. External traversal DSLs, however, lack support for incremental addition as they tend to generate code in bulk rather than small segments that can be integrated at a finer granularity. Therefore, programmers often face a *all-or-nothing* predicament, which limits their adoption. *Pure embedding* is a promising approach to address these limitations of external DSLs.

Other prominent research on traversal DSLs have focused on Strategic Programming (SP) [4, 11, 12] and Adaptive Programming (AP) [13, 14] paradigms, which support advanced separation of traversal concerns from type-specific actions. SP is a language-independent generic programming technique that provides a design method for programmer-definable, reusable, generic traversal schemes. AP, on the other hand, uses static meta-information to optimize traversals and check their conformance with the schema. Both the paradigms allow "structure-shy" programming to support traversal specifications that are loosely coupled to the object structure. We believe that the benefits of SP and AP are critical to the success of a traversal DSL. Therefore, an approach that combines

them in the context of a pure embedded DSL while addressing the *integration challenge* will have the highest potential for widespread adoption.

To address the limitations in the current state-of-the-art, in this paper we present a *generative programming* [15] -based approach to developing a pure embedded DSL for specifying traversals over object graphs governed by a schema. We present an expression-based [8] pure embedded DSL in C++ called **L**anguage for **E**mbedded qu**E**ry and traver**SA**l (LEESA), which leverages C++ templates and operator overloading to provide an intuitive notation for writing traversals. LEESA makes the following novel contributions:

- It provides a notation for traversal along several object structure axes, such as *child*, *parent*, *sibling*, *descendant*, and *ancestor*, which are akin to the XML programming idioms in XPath [16] – an XML query language. LEESA additionally allows composition of type-specific behavior over the axes-oriented traversals without tangling them together.
- It is a novel incarnation of SP using C++ templates, which provides a combinator style to develop programmer-definable, reusable, generic traversals akin to the classic SP language Stratego [17]. The novelty of LEESA's incarnation of SP stems from its use of static meta-information to implement not only the regular behavior of (some) primitive SP combinators but also their customizations to prevent traversals into unnecessary substructures. As a result, efficient *descendant* axis traversals are possible while simultaneously maintaining the schema-conformance aspect of AP.
- One of the most vexing issues in embedded implementations of DSLs is the lack of mechanisms for intuitive *domain-specific error reporting*. LEESA addresses this issue by combining C++ template metaprogramming [18] with concept checking [19, 20] in novel ways to provide intuitive error messages in terms of *concept violations* when incompatible traversals are composed at compile-time.
- Finally, its embedded approach allows incremental integration of the above capabilities into the existing code-base. During our evaluation of LEESA's capabilities, small segments of verbose traversal code were replaced by succinct LEESA expressions in a step by step fashion. We are not aware of any external C++ code generator that allows integration at comparable granularity and ease.

The remainder of the paper is organized as follows. Section 2 describes LEESA's notation for object structure traversal and its support for strategic programming; Section 3 presents how we have realized the capabilities of LEESA; Section 4 describes how concept checking is used in novel ways to provide domain-specific error diagnostics; Section 5 evaluates the effectiveness of LEESA; Section 6 and Section 7 present related work and conclusions, respectively.

2 The LEESA Language Design

In this section we formally describe the syntax of LEESA and its underlying semantic model in terms of axes traversals. To better explain its syntax and

semantics, we use a running example of a domain-specific modeling language for hierarchical finite state machine (HFSM) described next.

2.1 Hierarchical Finite State Machine (HFSM) Language: A Case-Study

Figure 1 shows a metamodel of a HFSM language using a UML-like notation. Our HFSM metamodel consists of `StateMachine`s with zero or more `State`s having directional edges between them called `Transition`s. States can be marked as a "start state" using a boolean attribute. States may contain other states, transitions, and optionally a `Time` element. 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 1. `Time` is an indivisible modeling element (hence the stereotype ≪Atom≫), which represents a user-definable delay in seconds. If it is absent, a default delay of 1 second is assumed. `Delay` represents the composition role of a `Time` object within a `State` object. All other composition relationships do not have any user-defined composition roles but rather a default role is assumed. The `Root` is a singleton that represents the root level model.

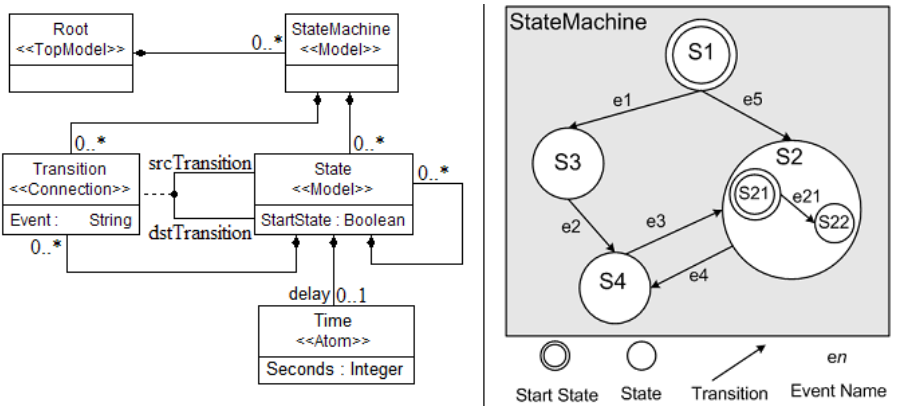

Fig. 1. Meta-model of Hierarchical Finite State Machine (HFSM) language (left) and a simple HFSM model (right)

To manipulate the instances of the HFSM language, C++ language bindings were obtained using a code generator. The generated code consists of five C++ classes: `Root`, `StateMachine`, `State`, `Transition`, and `Time` that capture the vocabulary and the relationships shown in the above metamodel. We use these classes throughout the paper to specify traversals listings.

2.2 An Axes-Oriented Notation for Object Structure Traversal

Designing an intuitive domain-specific notation for a DSL is central to achieving productivity improvements as domain-specific notations 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 DSL programs become more readable and maintainable than the programs written in general-purpose programming languages. For object structure traversal, the key abstractions are the objects and their typed collections while the basic operations performed are the navigation of associations and execution of type-specific actions.

When designing a notation for an embedded DSL, an important constraint imposed by the host language is to remain within the limits of the programmer-definable operator syntax offered by the host language. Quite often, trade-offs must be made to seek a balance between the expressiveness of the embedded notation against what is possible in a host language.

```
Statement   : Type { (Operator    Type)  |  (LRShift  visitor-object) |
                     (LRShift   Action)  |  (>>  Members)             |
                     (>> | >>=  Association)  }+
Type        : class-name '('  ')'
Operator    : LRShift |  >>=  |  <<=
LRShift     : >>  |  <<
Action      : "Select" | "Sort" | "Unique" | "ForEach" | (and more ...)
Association : "Association" '(' class-name :: role-name ')'
Members     : "MembersOf" '(' Type {  ','  Statement }+ ')'
```

Listing 1. Grammar of LEESA expressions

Listing 1 shows LEESA's syntax represented in the form of a grammar. *Statement* marks the beginning of a LEESA expression, which usually contains a series of *type*s, *action*s, and visitor objects separated by *operator*s. The first *type* in a LEESA statement determines the type of object where the traversal should begin. The four operators ($\gg$, $\ll$, $\gg=$, $\ll=$) are used to choose between *children* and *parent* axes and variations thereof. This traversal notation of LEESA resembles XPath's query syntax, however, unlike XPath, the LEESA expressions can be decorated with visitor objects, which modularize the type-specific actions away from the traversals.

The *association* production in Listing 1 represents traversal along user-defined association roles (captured in the metamodel) whereas *members* represent traversal along *sibling* axis. Actions are generic functions used to process the results of intermediate traversals. The parameters accepted by these actions, which are implemented as C++ function templates, are not shown. Instead, only the string literals sufficient for illustration are shown. Finally, instances of programmer-defined visitor classes can be added in the place of *visitor-object* that simply dispatch the type-specific actions. It conveniently allows accumulation of information during traversal without tangling the type-specific computations and the traversal specifications.

We now present concrete examples of LEESA expressions with their semantics in the context of the HFSM language case-study in Section 2.1.

Table 1. Child and parent axes traversal using LEESA (v can be replaced by an instance of a programmer-defined visitor class)

Axes	LEESA expressions and their semantics
(A) Child (breadth first)	**Root() ≫ StateMachine() ≫ v ≫ State() ≫ v** Visit *all* state machines followed by all their immediate children states.
(B) Child (depth first)	**Root() ≫= StateMachine() ≫ v ≫= State() ≫ v** Visit *a* state machine and all its immediate children states. Repeat this for the remaining state machines.
(C) Parent (breadth first)	**Time() ≪ v ≪ State() ≪ v ≪ StateMachine() ≪ v** Visit a given set of time objects followed by their immediate parent states followed by their immediate parent state machines.
(D) Parent (depth first)	**Time() ≪ v ≪= State() ≪ v ≪= StateMachine() ≪ v** For a given set of time objects, visit *a* Time object followed by visit its parent state followed by visit its parent state machine. Repeat this for the remaining time objects.

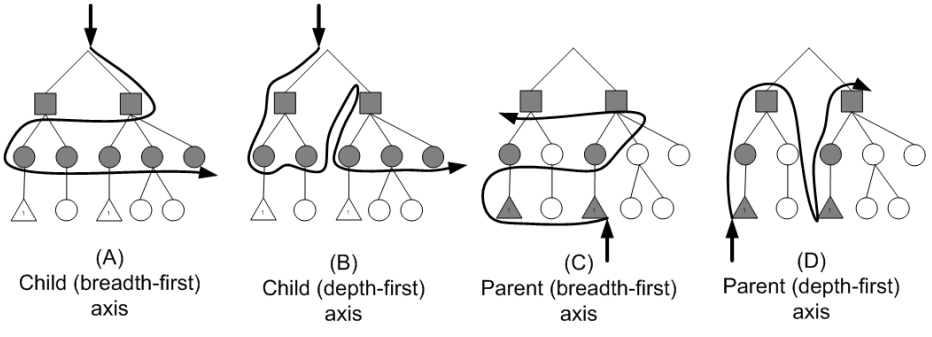

Fig. 2. Outlines of child/parent axes traversals (Squares are statemachines, circles are states, triangles are time objects, and shaded shapes are visited)

Child and Parent Axes. Child and parent axes traversals are one of the most common operations performed on object structures. LEESA provides a succinct and expressive syntax in terms of "≫" and "≪" operators for child and parent axes traversals, respectively. Two variations, *breadth-first* and *depth-first*, of both the axes are also supported. Presence of the "=" operator after the above operators turns a breadth-first strategy into a depth-first.[1] Table 1 shows four LEESA traversal expressions using child and parent axes notations. Figure 2 illustrates the graphical outlines corresponding to the examples shown in Table 1.

[1] In C++, "≪=" and "≫=" are bitwise shift left & assign and shift right & assign operators, respectively.

Breadth-first and depth-first variations of the axes traversal strategies are of particular interest here because of the ease of control over traversal provided by them. The breadth-first strategy, if applied successively (as in examples (a) and (c) in Table 1), visits all the instances of the specified type in a *group* before moving on to the next group of objects along an axis. Essentially, this strategy simulates multiple looping constructs in a sequence. The depth-first strategy, on the other hand, selects a single object of the specified type at a time, descends into it, executes the remaining traversal expression in the context of that single object, and repeats the same with the next object, if any. Therefore, successive application of the depth-first strategy (as in examples (b) and (d) in Table 1), traverses the edges of the object tree unlike the breadth-first strategy. Essentially, the depth-first strategy simulates nested looping constructs.

LEESA uses the Visitor [3] design pattern to organize the type-specific behavior while restricting traversals to the LEESA expressions only. To invoke type-specific computations, LEESA expressions can be decorated with instances of programmer-defined visitor classes as shown in Table 1. If a visitor object `v` is written after type `T`, LEESA invokes `v.Visit(t)` function on every collected object `t` of type `T`. LEESA expressions can be used not only for visitor dispatch but also for obtaining a collection of the objects of the type that appears last in the expression. Such a collection of objects can be processed using conventional C++. For instance, example (a) in Table 1 returns a set of `State`s whereas example (c) returns a set of `StateMachine`s.

Descendant and Ancestor Axes. LEESA supports *descendant* and *ancestor* axes traversal seamlessly in conjunction with child/parent axes traversals. For instance, Listing 2 shows a LEESA expression to obtain a set of `Time` objects that are recursively contained inside a `StateMachine`. This expression supports a form of structure-shy traversal in the sense that it does not explicitly specify the intermediate structural elements between the `StateMachine` and `Time`.

```
Root() >> StateMachine() >> DescendantsOf(StateMachine(), Time())
```

Listing 2. A LEESA expression showing descendant axis traversal

Two important issues arise in the design and implementation of the structure-shy traversal support described above. First, how are the objects of the target type located *efficiently* in a hierarchical object structure? and second, at what stage of development the programmer is notified of impossible traversal specifications? In the first case, for instance, it is inefficient to search for objects of the target type in composites that do not contain them. Whereas in the second case, it is erroneous to specify a target type that is not reachable from the start type. Section 3.3 and Section 4 present solutions to the efficiency and the error reporting issues, respectively.

Sibling Axis. Composition of multiple types of objects in a composite object is commonly observed in practice. For example, the HFSM language has a composite called `StateMachine` that consists of two types of children that are siblings

of each other: `State` and `Transition`. Support for object structure traversal in LEESA would not be complete unless support is provided for visiting multiple types of siblings in a programmer-defined order.

```
ProgrammerDefinedVisitor v;
Root() >> StateMachine() >> MembersOf(StateMachine(), State()      >> v,
                                                       Transition() >> v)
```

Listing 3. A LEESA expression for traversing siblings: `States` and `Transitions`

Listing 3 shows an example of how LEESA supports sibling traversal. The sample expression visits all the `State`s in a `StateMachine` before all the `Transitions`. The types of visited siblings and their order is programmer-definable. The `MembersOf` notation is designed to improve readability as its first parameter is the common parent type (*i.e.,* `StateMachine`) followed by a comma separated list of LEESA subexpressions for visiting the children in the given order. It effectively replaces multiple *for* loops written in a sequence where each for loop corresponds to a type of sibling.

Association Axis. LEESA supports traversals along two different kinds of user-defined associations. First, *named composition roles*, which use user-defined roles while traversing composition instead of the default composition role. For instance, in our HFSM modeling language, `Time` objects are composed using the `delay` composition role inside `State`s. Second, *named associations* between different types of objects that turn tree-like object structures into graphs. For example, `Transition` is a user-defined association possible between any two `State`s in the HFSM language described in Section 2.1. Moreover, `srcTransition` and `dstTransition` are two possible roles a `State` can be in with respect to a `Transition`.

LEESA provides a notation to traverse an association using the name of the association class (*i.e., class-name* in Listing 1) and the desired role (*i.e., role-name* in Listing 1). Listing 4 shows two independent LEESA expressions that traverse two different user-defined associations. The first expression returns a set of `Time` objects that are composed immediately inside the top-level `State`s. The expression traverses the `delay` composition role defined between states and time objects. This feature allows differentiation (and selection) of children objects that are of the same type but associated with their parent with different composition roles.

```
Root() >> StateMachine() >> State() >> Association(State::delay)  ...(1)
Root() >> StateMachine() >> Transition()
                         >> Association(Transition::dstTransition)  ...(2)
```

Listing 4. Traversing user-defined associations using LEESA

The second expression returns all the top-level states that have at least one incoming transition. Such a set can be conceptually visualized as a set of states

that are at the *destination* end of a transition. The second expression in Listing 4 up to `Transition()` yields a set of transitions that are the immediate children of `StateMachine`s. The remaining expression to the right of it traverses the user-defined association `dstTransition` and returns `State`s that are in the *destination* role with respect to every `Transition` in the previously obtained set.

In the above association-based traversals, the operator "≫" does not imply child axis traversal but instead represents continuation of the LEESA expression in a breadth-first manner. As described before, breadth-first strategy simulates loops in sequence. Use of "≫=" turns the breadth-first strategy over association axis into a depth-first strategy, which simulates nested loops. Expressions with associations can also be combined with visitor objects if role-specific actions are to be dispatched.

2.3 Programmer-Defined Processing of Intermediate Results Using Actions

Writing traversals over object structures often requires processing the intermediate results before the rest of the traversal is executed (*e.g.,* filtering objects that do not satisfy a programmer-defined predicate, or sorting objects using programmer-defined comparison functions). LEESA provides a set of *actions* that process the intermediate results produced by the earlier part of the traversal expression. These actions are in fact higher-order functions that take programmer-defined predicates or comparison functions as parameters and apply them on a collection of objects.

```
int comparator (State, State) { ... } // A C++ comparator function
bool predicate (Time) { ... }         // A C++ predicate function
Root() >> StateMachine() >> State() >> Sort(State(), comparator)
                         >> Time()  >> Select(Time(), predicate)
```

Listing 5. A LEESA expression with actions to process intermediate results

Listing 5 shows a LEESA expression that uses two predefined actions: `Sort` and `Select`. The `Sort` function, as the name suggests, sorts a collection using a programmer-defined comparator. `Select` filters out objects that do not satisfy the programmer-defined predicate. The result of the traversal in Listing 5 is a set of `Time` objects, however, the intermediate results are processed by the actions before traversing composition relationships further. `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*.

LEESA supports about a dozen different actions (*e.g.,* Unique, ForEach) and more actions can be defined by the programmers and incorporated into LEESA expressions if needed. The efforts needed to add a new action are proportional to adding a new class template and a global overloaded operator function template.

2.4 Generic, Recursive, and Reusable Traversals Using Strategic Programming

Although LEESA's axes traversal operators ($\gg$, $\ll$, $\gg=$, $\ll=$) are reusable for writing traversals across different schemas, they force the programmers to commit to the vocabulary of the schema and therefore the traversal expressions (as whole) cannot be reused. Moreover, LEESA's axes traversal notation discussed so far lacked support for *recursive* traversal, which is important for a wide spectrum of domain-specific modeling languages that support *hierarchical* constructs. For example, our case study of HFSM modeling language requires recursive traversal to visit deeply nested states.

A desirable solution should not only support recursive traversals but also enable higher-level reuse of *traversal schemes* while providing complete control over traversal. Traversal schemes are higher level control patterns (*e.g.,* top-down, bottom-up, depth-first, etc.) for traversal over heterogeneously typed object structures. Strategic Programming (SP) [4, 11, 12] is a well known generic programming idiom based on programmer-definable (recursive or otherwise) traversal abstractions that allow separation of type-specific actions from reusable traversal schemes. SP also provides a *design method* for developing reusable traversal functionality based on so called *strategies*. Therefore, based on the observation that LEESA shares this goal with that of SP, we adopted the SP design method and created a new incarnation of SP on top of LEESA's axes traversal notation. Next, we describe how LEESA leverages the SP design method to meet its goal of supporting generic, recursive, and reusable traversals. For a detailed description of the foundations of SP, we suggest reading [4, 11, 12] to the readers.

LEESA's incarnation of the SP design method is based on a small set of *combinators* that can be used to construct new combinators from the given ones. By combinators we mean reusable C++ class templates capturing basic functionality that can be composed in different ways to obtain new functionality. The basic combinators supported in LEESA are summarized in Table 2. This set of combinators is inspired by the *strategy* primitives of the term rewriting language Stratego [17].

Table 2. The set of basic class template combinators

Primitive combinators	**Description**
Identity	Returns its input datum without change.
Fail	Always throws an exception indicating a failure.
Sequence<S1,S2>	Invokes strategies S1 and S2 in sequence on its input datum.
Choice<S1,S2>	Invokes strategy S2 on its input datum only if the invocation of S1 fails.
All<S>	Invokes strategy S on all the immediate children of its input datum.
One<S>	Stops invocation of strategy S after its first success on one of the children of its input datum.

```
FullTD<Strategy> = Sequence<Strategy, All<FullTD> >
FullBU<Strategy> = Sequence<All<FullBU>, Strategy>
```

Listing 6. Pseudo-definitions of the class templates of the predefined traversal schemes (Strategy = Any primitive combinator or combination thereof, TD = top-down, BU = bottom-up)

`All` and `One` are *one-layer traversal* combinators, which can be used to obtain full traversal control, including recursion. Although none of the basic combinators are recursive, higher-level traversal schemes built using the basic combinators can be recursive. For instance, Listing 6 shows a subset of predefined higher-level traversal schemes in LEESA that are recursive. The (pseudo-) definition of `FullTD` (full top-down) means that the parameter `Strategy` is applied at the root of the incoming datum and then it applies itself recursively to all the immediate children of the root, which can be of heterogeneous types. Figure 3 shows a graphical illustration of `FullTD` and `FullBU` (full bottom-up) traversal schemes. Section 3.3 describes the actual C++ implementation of the primitives and the recursive schemes in detail.

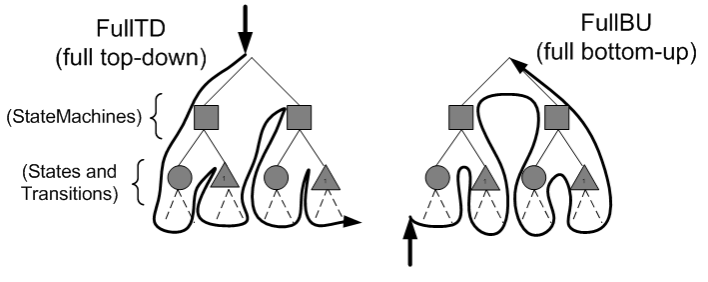

Fig. 3. Graphical illustration of FullTD and FullBU traversal schemes. (Squares, circles, and triangles represent objects of different types).

```
Root() >> StateMachine() >> FullTD(StateMachine(), VisitStrategy(v));
```

Listing 7. Combining axes traversal with strategic programming in LEESA. (v can be replaced by a programmer-defined visitor.)

Listing 7 shows how `FullTD` recursive traversal scheme can be used to perform full top-down traversal starting from a `StateMachine`. Note that the heterogeneously typed substructures (`State`, `Transition`, and `Time`) of the `StateMachine` are not mentioned in the expression. However, they are incorporated in the traversal automatically using the static meta-information in the metamodel. This is achieved by *externalizing* the static meta-information in a form that is understood by the C++ compiler and in turn the LEESA expressions. Later in Section 3.2 we describe a process of externalizing the static meta-information

from the metamodel (schema) and in Section 3.3 we show how it is used for substructure traversal.

Finally, the `VisitStrategy` in Listing 7 is a predefined LEESA strategy that can not only be configured with programmer-defined visitor objects but can also be replaced by other programmer-defined strategies. We envision that LEESA's `VisitStrategy` will be used predominantly because it supports the *hierarchical visitor* pattern [21] to keep track of depth during traversal. This pattern is based on a pair of type-specific actions: `Visit` and `Leave`. The prior one is invoked while *entering* a non-leaf node and the latter one is invoked while *leaving* it. To keep track of depth, the visitor typically maintains an internal stack where the `Visit` function does a "push" operation and `Leave` function does a "pop".

2.5 Schema Compatibility Checking

Every syntactically correct traversal expression in LEESA is statically checked against the schema for type errors and any violations are reported back to the programmer. Broadly, LEESA supports four kinds of checks based on the types and actions participating in the expression. First, only the types representing the vocabulary of the schema are allowed in a LEESA expression. The visitor instances are an exception to this rule. Second, impossible traversal specifications are rejected where there is no way of reaching the elements of a specified type along the axis used in the expression. For example, the child-axis operators ($\gg$, $\gg=$) require (immediate) parent/child relationship between the participating types whereas `DescendantsOf` requires a transitive closure of the child relationship. Third, the argument type of the intermediate results processing actions must match to that of the result returned by the previous expression. Finally, the result type of the action must be a type from the schema if the expression is continued further. Section 4 describes in detail how we have implemented schema compatibility checking using C++ Concepts.

3 The Implementation of LEESA

In this section we present LEESA's layered software architecture, the software process of obtaining the static meta-information from the schema, and how we have implemented the strategic traversal combinators in LEESA.

3.1 The Layered Architecture of LEESA

Figure 4 shows LEESA's layered architecture. At the bottom is the in-memory *object structure*, which could be a tree or a graph. An *object-oriented data access layer* is a layer of abstraction over the object structure, which provides schema-specific, type-safe interfaces for iteratively accessing the elements in the object structure. Often, a code generator is used to generate language bindings (usually a set of classes) that model the vocabulary. Several different types of code generators such as XML schema compilers [22] and domain-specific modeling

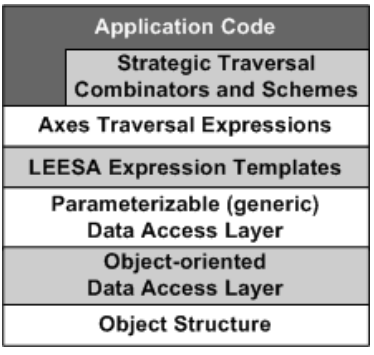

Fig. 4. Layered View of LEESA's Architecture (Shading of blocks shown for aesthetic reasons only)

tool-suites [23] are available that generate schema-specific object-oriented data access layer from the static meta-information.

To support generic traversals, the schema-specific object-oriented data access layer must be adapted to make it suitable to work with the generic implementation of LEESA's C++ templates. The *parameterizable data access layer* is a thin generic wrapper that achieves this. It treats the classes that model the vocabulary as type parameters and hides the schema-specific interfaces of the classes. This layer exposes a small generic interface, say, `getChildren`, to obtain the children of a specific type from a composite object and say, `getParent`, to obtain the parent of an object. For example, using C++ templates, obtaining children of type `T` of an object of type `U` could be implemented[2] as `U.getChildren<T>()`, where `U` and `T` could be any two classes modeling the vocabulary that have parent/child relationship. This layer can also be generated automatically from the object structure schema.

Expression Templates [24] is the key idea behind embedding LEESA's traversal expressions in C++. Using operator overloading, expression templates enable *lazy evaluation* of C++ expressions, which is otherwise not supported natively in C++. Lazy evaluation allows expressions – rather than their results – to be passed as arguments to functions to extract results lazily when needed. LEESA overloads the $\gg$, $\ll$, $\gg=$, and $\ll=$ operators using the design method of expression templates to give embedded traversal expressions a look and feel of XPath's axes-oriented traversal specifications. Moreover, LEESA expressions can be passed to other generic functions as arguments to extract results lazily. LEESA's expression templates map the traversal expressions embedded in a C++ program onto the parameterizable data access layer. They raise the level of abstraction by hiding away the iterative process of accessing objects and instead focus only on the *relevant* types in the vocabulary and different strategies (breadth-first and depth-first) of traversal. LEESA's expression templates

[2] A widely supported, standard C++ feature called "template explicit specialization" could be used.

are independent of the underlying vocabulary. Schema-specific traversals are obtained by instantiating them with schema-specific classes. For more details on LEESA's expression templates, including an example, the readers are directed to our previous work [25].

Finally, LEESA programmers use the *axes traversal expressions* and *strategic traversal combinators and schemes* to write their traversals as described in Section 2. The axes traversal expressions are based on LEESA's expression templates. The strategic traversal combinators use an externalized representation of the static meta-information for their generic implementation. Below we describe the process of externalizing the static meta-information.

3.2 Externalizing Static Meta-information

Figure 5 shows the software process of developing a schema-first application using LEESA. The object-oriented data access layer, parameterizable data access layer, and the static meta-information are generated from the schema using a code generator. Conventional [23, 22] code generators for language-specific bindings generate the object-oriented data access layer only, but for this paper we extended the Universal Data Model (UDM) [23] – a tool-suite for developing domain-specific modeling languages (DSML) – to generate the parameterizable data access layer and the static meta-information. The cost of extending UDM is amortized over the number of schema-first applications developed using LEESA. While the static meta-information is used for generic implementations of the primitive strategic combinators, C++ Concepts [19, 20] shown in Figure 5 are used to check the compatibility of LEESA expressions with the schema and report the errors back to the programmer at compile-time. C++ Concepts allow the error messages to be succinct and intuitive. Such a diagnosis capability is

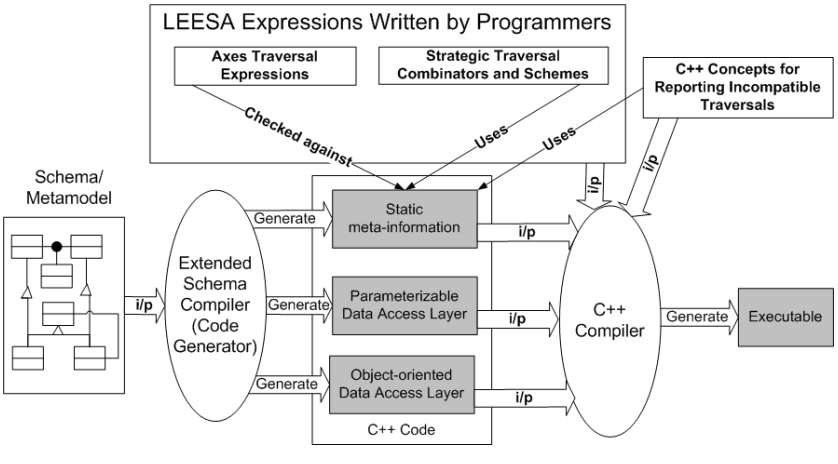

Fig. 5. The software process of developing a schema-first application using LEESA. (Ovals are tools whereas shaded rectangular blocks represent generated code).

of high practical importance as it catches programmer mistakes much earlier in the development lifecycle by providing an additional layer of safety.

The Boost C++ template metaprogramming library (MPL) [18] has been used as a vehicle to represent the static meta-information in LEESA. It provides easy to use, readable, and portable mechanisms for implementing metaprograms in C++. MPL has become a de-facto standard for metaprogramming in C++ with a collection of extensible compile-time algorithms, typelists, and metafunctions. Typelists encapsulate zero or more C++ types (programmer-defined or otherwise) in a way that can be manipulated at compile-time using MPL metafunctions.

Using Boost MPL to Externalize the Static Meta-information. The static meta-information (partial) of the HFSM metamodel (Section 2.1) captured using Boost MPL typelists is shown below.

```
class StateMachine {
   typedef mpl::vector < State, Transition > Children;
};
class State {
   typedef mpl::vector < State, Transition, Time > Children;
};
class Transition {                          // Same as class Time
   typedef mpl::vector < > Children         // empty
};
mpl::contains  <StateMachine::Children, State>::value //...(1) true
mpl::front     <State::Children>::type                //...(2) class State
mpl::pop_front <State::Children>::type                //...(3) mpl::vector<Transition, Time>
```

Each class has an associated type called `Children`, which is a MPL typelist implemented using `mpl::vector`. The typelist contains a list of types that are children of its host type. A MPL metafunction called `mpl::contains` has been used to check existence of a type in a MPL typelist. For example, the statement indicated by (1) above checks whether typelist `StateMachine::Children` contains type `State` in it or not. It results in a compile-time constant *true* value. Metafunctions `mpl::front` and `mpl::pop_front`, indicated by (2) and (3), are semantically equivalent to "car" and "cdr" list manipulation functions in Lisp. While `mpl::front` returns the first type in the typelist, `mpl::pop_front` removes the first type and returns the remaining typelist.

We leverage this metaprogramming support provided by MPL to represent children, parent, and descendant axes meta-information in C++. We have extended the UDM tool-suite to generate Boost MPL typelists that capture the static meta-information of these axes.

3.3 The Implementation of Strategic Traversal Schemes

In LEESA's implementation of SP, `All` and `One` are *generative* one-layer combinators because their use requires mentioning the type of only the start element where the strategy application begins. The children and descendant (in case of recursive traversal schemes) types of the start type are automatically incorporated into the traversal using the externalized static meta-information and the LEESA's metaprograms that iterate over it.

```
template <class Strategy>
class All {
      Strategy strategy_;
   public:
      All (Strategy s) : strategy_(s) { }    // Constructor
      template <class T>
      void apply (T arg) {    // Every strategy implements this member template function.
          // If T::Children typelist is empty, calls (B) otherwise calls (A)
          children(arg, typename T::Children());
      }
   private:
      template <class T, class Children>
      void children(T arg, Children)   {                          // ............ (A)
         typedef typename mpl::front<Children>::type Head;        // ...(1)
         typedef typename mpl::pop_front<Children>::type Tail;    // ...(2)
         for_each c in arg.getChildren<Head>()                    // ...(3)
             strategy_.apply(c);
         children(arg, Tail());                                   // ...(4)
      }
      template <class T>
      void children(T, mpl::vector<> /* empty typelist */) { }    // ............ (B)
};
```

```
template <class S1, class S2>
class SEQ {
   S1 s1_; S2 s2_;
public:
   SEQ(S1 s1, S2 s2)
      : s1_(s1), s2_(s2) {}
   template <class T>
   void apply (T arg) {
      s1_.apply(arg);
      s2_.apply(arg);
   }
};
```

```
template <class Strategy>
class FullTD {
    Strategy st_;
public:
    FullTD(Strategy s) : st_(s) {}
    template <class T>
    void apply (T arg) {
       All<FullTD> all(*this);
       SEQ<Strategy, All<FullTD> > seq(st_, all);
       seq.apply(arg);
    }
};
```

Listing 8. C++ implementations of `All` and `SEQ` (Sequence) primitive combinators and the `FullTD` recursive traversal scheme

Listing 8 shows the C++ implementation of the `All` and `Sequence` primitive combinators and the `FullTD` recursive traversal scheme in LEESA. `All` is a class template that accepts `Strategy` as a type parameter, which could be instantiations of other combinators or other instantiations of `All` itself. Execution of `All` begins at the `apply` function, which delegates execution to another member template function called `children`. `All::children` is instantiated as many times as there are children of type `T`. From the `T::Children` typelist, repeated instantiation of the `children` member template function are obtained using the metaprogram indicated by statements (1), (2), and (4) in Listing 8.

Similar to list processing in functional languages, statement (1) yields the first type (`Head`) in the typelist whereas statement (2) yields the remaining typelist (`Tail`). Statement (4) is a compile-time recursive call to itself but with `Tail` as its second parameter. This compile-time recursion terminates only when `Tail` becomes empty after successive application of `mpl::pop_front` metafunction. When `Tail` is an empty typelist, `children` function marked by (B) is invoked terminating the compile-time recursion. Figure 6 shows graphical illustration of this recursive instantiation process. Multiple recursive instantiations of function `children` are shown in the order they are created with progressively smaller and

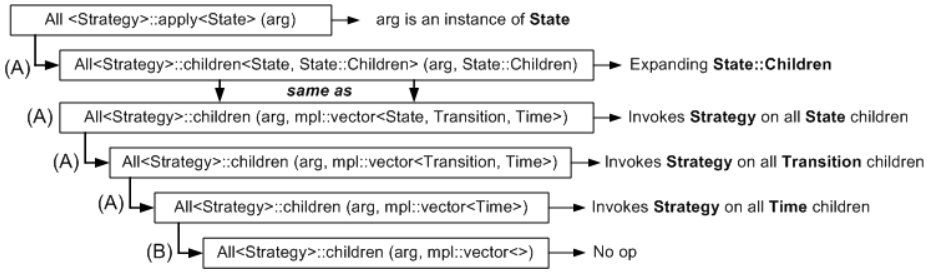

Fig. 6. Compile-time recursive instantiations of the `children` function starting at `All<Strategy>::apply<State>(arg)` when arg is of type `State`

smaller typelist as its second parameter. Finally, the statement marked as (3) is using the parameterizable data access interface `T::getChildren`, which returns all the `Head` type children of `arg`.

Efficient Descendant Axis Traversal. Compile-time customizations of the primitive combinator `All` and in turn `FullTD` traversal scheme are used for efficient implementation of the *descendant* axis traversal. LEESA can prevent traversals into unnecessary substructures by controlling the types that are visited during recursive traversal of `FullTD` and `FullBU` schemes. LEESA customizes the behavior of the `All` primitive combinator using the descendant types information that is obtained from the schema. The `T::Children` typelist in the `All::apply` function is manipulated using C++ template metaprogramming such that the schema types that have no way of reaching the objects of the target type are eliminated before invoking the `All::children` function. This is achieved using compile-time boolean queries over the list of descendant types implemented using MPL's metafunction `mpl::contains` as described in Section 3.2. All these metaprograms are completely encapsulated inside the C++ class templates that implement recursive traversal schemes and are not exposed to the programmers.

While static meta-information can be used for efficient traversal, the same meta-information can be used to check the LEESA expressions for their compatibility with the schema. We describe that next.

4 Domain-Specific Error Reporting Using C++ Concepts

In DSL literature [7, 8], embedded DSLs have been criticized for their lack of support for domain-specific error reporting. The importance of intuitive error messages should not be underestimated as it directly affects the programmer's effectiveness in locating and correcting errors in a DSL program. This issue is all the more important for embedded DSLs since their compiler is the same as the host language compiler, and hence the error reports are often in terms of the host language artifacts instead of domain-specific artifacts that are relevant to the problem. Moreover, for embedded DSLs in C++ that are implemented

using templates, the problem is further exacerbated because templates lack early modular (separate) type-checking.

4.1 Early Type-Checking of C++ Templates Using Concepts

C++ Concepts [19] have been proposed in the latest C++ programming language standard, C++0x [26], to address the problem of late type-checking of templates during compilation. Concepts express the syntactic and semantic behavior of types and constrain the type parameters in a C++ template, which are otherwise unconstrained. Concepts allow separate type-checking of template definitions from their uses, which makes templates easier to use and easier to compile. The set of constraints on one or more types are referred to as *Concepts*. Concepts describe not only the functions and operators that the types must support but also other accessible types called *associated types*. The types that satisfy the requirements of a concept are said to *model* that concept. When a concept constrained C++ template is instantiated with a type that does not model the concept, an error message indicating the failure of the concept and the type that violates it are shown at the call site in the source code. An experimental support for C++ Concepts has been implemented in the ConceptGCC [27] compiler[3].

4.2 Schema Compatibility Checking Using Concepts and Metaprogramming

We have defined several C++ Concepts in LEESA that must be satisfied by different types participating in a LEESA expression. These Concepts are related primarily to child, parent, descendant, and ancestor axes traversals and the invocation of actions for intermediate results processing. For example, each type in a child axis traversal expression must model a `ParentChildConcept` with respect to its preceding type. An implementation of the `ParentChildConcept` is shown below.

```
concept ParentChildConcept <typename Parent, typename Child> {
  typename Children = typename Parent::Children;
  typename IsChild = typename mpl::contains<Children, Child>::type;
  requires std::SameType<IsChild, true_type>;
};
```

The concept is parameterized with two types and essentially requires that the `Child` type be present in the list of children of the `Parent` type. This is achieved by (1) obtaining the result of the application of MPL metafunction `contains` on `Parent`'s associated type `Children` and (2) enforcing the type of the result to be the same as `true_type`, which signifies success. If the Concept does not hold, a short error message is produced stating the failure of the Concept and the types that violate it. The error is reported at the first occurrence of the type that violates it regardless of the length of the expression. As the length of

[3] Library level support for concept checking is available [20] for pre-C++0x compilers.

the erroneous LEESA expression grows, the error output grows linearly due to increasing size of the recursively constructed type using expression templates. However, the reason and location are always stated distinctly in the form of concept violation.

For example, consider a LEESA expression, "`StateMachine()` ≫ `Time()`", which is incorrect with respect to the metamodel of the HFSM modeling language because `Time` is not an immediate child of `StateMachine` and therefore, does not satisfy the `ParentChildConcept` described before. Below, we have shown the actual error message produced by the ConceptGCC [27] compiler, which is only four lines long, and clearly states the reason and the location (both on the fourth line) of the error.

```
t.cc: In function 'int main()':
t.cc:99: error: no match for 'operator>>' in 'StateMachine() >> Time()'
t.cc:85: note: candidates are: R LEESA::operator>>(const L&, const R&)
               [with L = StateMachine, R = Time] <requirements>
t.cc:99: note: no concept map for requirement
         'LEESA::ParentChildConcept<StateMachine, Time>'
```

5 Evaluation

In this section, we present quantitative results on LEESA's effectiveness in reducing efforts while programming traversals compared to the third generation object-oriented languages.

Experimental setup. We conducted the experiments using our open-source domain-specific modeling tool-suite: CoSMIC.[4] CoSMIC is a collection of domain-specific modeling languages (DSML), interpreters, code generators, and model-to-model transformations developed using Generic Modeling Environment (GME) [28], which is a meta-programmable tool for developing DSMLs. CoSMIC's DSMLs are used for developing distributed applications based on component-based middleware. For instance, the Platform Independent Component Modeling Language (PICML) [29] is one of the largest DSMLs in CoSMIC for modeling key artifacts in all the life-cycle phases of a component-based application, such as interface specification, component implementation, hierarchical composition of components, and deployment. A PICML model may contain up to 300 different types of objects. Also, PICML has over a dozen model interpreters that generate XML descriptors pertaining to different application life-cycle stages. All these interpreters are implemented in C++ using UDM as the underlying object-oriented data access layer.

Methodology. The objective of our evaluation methodology is to show the reduction in programming efforts needed to implement commonly observed traversal patterns using LEESA over traditional iterative constructs.

To enable this comparison, we refactored and reimplemented the traversal related parts of PICML's deployment descriptor generator using LEESA. This generator exercises the widest variety of traversal patterns applicable to PICML.

[4] http://wwww.dre.vanderbilt.edu/cosmic

Table 3. Reduction in code size (# of lines) due to the replacement of common traversal patterns by LEESA expressions

Traversal Pattern	**Axis**	**Occurrences**	**Original #lines** (average)	**#Lines using LEESA** (average)
A single loop iterating over a list of objects	Child	11	8.45	1.45
	Association	6	7.50	1.33
5 sequential loops iterating over siblings	Sibling	3	41.33	6
2 Nested loops	Child	2	16	1
Traversal-only visit functions	Child	3	11	0
Leaf-node accumulation using depth-first	Descendant	2	43.5	4.5
Total traversal code	-	All	414 (absolute)	53 (absolute)

It amounts to little over 2,000 lines[5] of C++ source code (LOC) out of which 414 (about 21%) LOC perform traversals. It is organized using the Visitor [3] pattern where the accept methods in the object structure classes are non-iterating and the entire traversal logic along with the type-specific actions are encapsulated inside a monolithic visitor class. Table 3 shows the traversal patterns we identified in the generator. We replaced these patterns with their equivalent constructs in LEESA. This procedure required some refactoring of the original code.

Analysis of results. Table 3 shows a significant reduction in the code size due to LEESA's succinct traversal notation. As expected, the highest reduction (by ratio) in code size was observed when two ad-hoc implementations of depth-first search (*e.g.,* searching nested components in a hierarchical component assembly) were replaced by LEESA's adaptive expressions traversing the descendant axis. However, the highest number of reduction in terms of the absolute LOC (114 lines) was observed in the frequently occurring traversal pattern of a *single loop*. Cumulatively, leveraging LEESA resulted in 87.2% reduction in traversal code in the deployment descriptor generator. We expect similar results in other applications of LEESA.

Incremental Adoption of LEESA. It is worth noting here that due to its pure embedded approach, applying LEESA in the existing model traversal programs is considerably simpler than external DSLs that generate code in bulk. Incremental refactoring of the original code-base was possible by replacing one traversal pattern at a time while being confident that the replacement is not changing the behavior in any unexpected ways. Such incremental refactoring using *external* traversal DSLs that use a code generator would be extremely hard, if not impossible. Our pure embedded DSL approach in LEESA allows us to distance ourselves from such *all-or-nothing* predicament, which could

[5] The number of lines of source code is measured excluding comments and blank lines.

potentially be a serious practical limitation. We expect that a large number of existing C++ applications that use XML data-binding [22] can start benefiting from LEESA using this incremental approach provided their XML schema compilers are extended to generate the parameterizable data access layer and the meta-information.

6 Related Work

In this section we place LEESA in the context of a sampling of the most relevant research efforts in the area of object structure traversal.

XPath 2.0 [16] is a structure-shy XML query language that allows node selection in a XML document using downward (children, descendant), upward (parent, ancestor), and sideways (sibling) axes. In general, XPath supports more powerful node selection expressions than LEESA using its untyped unconstrained (*i.e. axis::**) axes. XPath's formal semantics describe how XML schema could be used for *static type analysis* to detect certain type errors and to perform optimizations. However, contemporary XPath programming APIs for C++ use string encoded expressions, which are not checked against the schema at compile-time. Moreover, unlike XPath, type-specific behavior can be composed over the axes-oriented traversals using LEESA.

Adaptive Programming (AP) [13,14] specifies structure-shy traversals in terms of milestone classes composed using predicates, such as *from*, *to*, *through*, and *bypass*. It uses static meta-information to optimize traversals as well as to check their compatibility against the schema. While LEESA focuses on accumulation of nodes using its axes-oriented notation and programmability of traversals using its strategic combinator style, AP focuses on collection of unique paths in the object graph specified using the above mentioned predicates. The use of visitors in LEESA to modularize type-specific actions is similar in spirit to the code wrappers in AP.

Strategic Programming (SP) [4, 17], which began as a term rewriting [17] language has evolved into a language-interparadigmatic style of programming traversals and has been incarnated in several other contexts, such as functional [12], object-oriented [11], and embedded [10]. The strategic traversal expressions in LEESA are based on a new embedded incarnation of SP in an *imperative* language, C++. Unlike [10], however, no compiler extension is necessary. Also, all the expressions are *statically* checked against the schema, unlike visitor combinators [11].

Scrap++ [30] presents a C++ templates-based approach for implementing Haskell's "Scrap Your Boilerplate" (SYB) design pattern, which is remarkably similar to SP. Scrap++'s approach depends on recursive traversal combinators, a one-layer traversal, and a type extension of the basic computations. However, LEESA's approach is different in many significant ways. First, unlike LEESA, the objective of Scrap++ is to mimic Haskell's SYB and therefore does not provide an intuitive axes traversal notation. Second, LEESA presents a software process for generating schema-specific meta-information that is used during compilation

for generating traversals as well as compatibility checking. Third, SYB lacks parental and sibling contexts. Finally, no technique is provided in Scrap++ to produce intuitive error messages.

Lämmel et al. [31] present a way of realizing adaptive programming predicates (*e.g., from*, *to*, *through*, and *bypass*) by composing SP primitive combinators and traversal schemes. Due to a lack of static type information, their simulation of AP in terms of SP lacks important aspects of AP, such as static checking and avoiding unnecessary traversal into substructures. LEESA, on the other hand, uses the externalized meta-information to not only statically check the traversals but also makes them efficient.

Static meta-information has also been exploited by Cunha et al. [32] in the transformation of structure-shy XPath and SP programs for statically optimizing them. Both the approaches eliminate unnecessary traversals into substructures, however, no transformation is necessary in the case of LEESA. Instead, the behaviors of `All` and `One` primitive combinators are customized at compile-time to improve efficiency. Moreover, LEESA's structure-shy traversals support mutually recursive types, unlike [32].

Lämmel [33] sketches an encoding of XPath-like axes (downward, upward, and sideways) using strategic function combinators in the SYB style. LEESA is similar to this work because both the approaches suggest an improvement of XPath-like set of axes with support for strategic, recursive traversal abstractions and provide a way of performing schema-conformance checking. The key differences are the improved efficiency of the *descendant* axis traversal in case of LEESA, its domain-specific error reporting capability, and its use of an imperative, object-oriented language as opposed to Haskell, which is a pure functional language.

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 basic computations. A code generator is used to transform the traversal specifications into imperative code based on the Visitor pattern. This approach is, however, heavyweight compared to the embedded approach because it incurs high cost of the development and maintenance of the language processor.

Language Integrated Query (LINQ) [34] is a Microsoft .NET technology that supports 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 to XSD" promises to add much needed typed XML programming support over its predecessor "LINQ to XML." LINQ, however, does not support strategic combinator style like LEESA. The Object Constraint Language (OCL) [35] is a declarative language for describing well-formedness constraints and traversals over object structures represented using UML class graphs. OCL, however, does not support *side-effects* (*i.e.*, object structure transformations are not possible).

Czarnecki et al. [8] compare staged interpreter techniques in MetaOCaml with the template-based techniques in Template Haskell and C++ to implement embedded DSLs. Two approaches – *type-driven* and *expression-driven* – of implementing an embedded DSL in C++ are presented. Within this context, our

previous work [25] presents LEESA's expression-driven pure embedding approach. Spirit[6] and Blitz++[7] are two other prominent examples of expression-driven embedded DSLs in C++ for recursive descent parsing and scientific computing, respectively. Although LEESA shares the implementation technique of expression templates with them, strategic and XPath-like axes-oriented traversals cannot be developed using Spirit or Blitz++.

7 Conclusion

In this paper we presented a case for *pure embedding* in C++ as an effective way of implementing a DSL particularly in the domain of object structure traversal where mature implementations of iterative data access layer abound. While many of the existing embedded DSLs perform poorly with respect to domain-specific error reporting, our novel approach of fusing together C++ Concepts and compile-time type manipulation using template metaprogramming allows us to report impossible traversals by terminating compilation with short and intuitive error messages. We believe this technique is applicable to other embedded DSLs in C++ irrespective of their domain.

To show the feasibility of our approach, we developed **L**anguage for **E**mbedded qu**E**ry and traver**SA**l (LEESA), which is a pure embedded DSL in C++ for object structure traversal. LEESA improves the modularity of the traversal programs by separating the knowledge of the object structure from the type-specific computations. LEESA adopts the strategic combinator style to provide a library of generic reusable traversal schemes. With an eye on "structure shyness", LEESA supports XPath-like traversal axes to focus only on the relevant types of richly structured data to shield the programs from the effects of schema evolution. Our contribution lies in combining these powerful traversal techniques without sacrificing efficiency and schema-conformance checking.

LEESA is available for download in open-source at `http://www.dre.vanderbilt.edu/cosmic`.

Acknowledgment. We are grateful to Endre Magyari for his implementation support to enhance the UDM code generator. We also thank the DSL'09 program committee and our shepherd Dr. Ralf Lämmel for providing detailed reviews.

References

1. Thompson, H.S., Beech, D., Maloney, M., Mendelsohn, N., et al.: XML Schema Part 1: Structures. W3C Recommendation (2001)
2. Sztipanovits, J., Karsai, G.: Model-Integrated Computing. IEEE Computer 30(4), 110–112 (1997)

[6] http://spirit.sourceforge.net
[7] http://www.oonumerics.org/blitz

3. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)
4. Lämmel, R., Visser, E., Visser, J.: The Essence of Strategic Programming, p. 18 (October 15, 2002), http://www.cwi.nl/~ralf
5. Ovlinger, J., Wand, M.: A Language for Specifying Recursive Traversals of Object Structures. SIGPLAN Not. 34(10), 70–81 (1999)
6. Gray, J., Karsai, G.: An Examination of DSLs for Concisely Representing Model Traversals and Transformations. In: 36th Hawaiian International Conference on System Sciences (HICSS), pp. 325–334 (2003)
7. Mernik, M., Heering, J., Sloane, A.M.: When and How to Develop Domain-specific Languages. ACM Computing Surveys 37(4), 316–344 (2005)
8. Czarnecki, K., O'Donnell, J., Striegnitz, J., Taha, W.: DSL Implementation in MetaOCaml, Template Haskell, and C++. In: Domain Specific Program Generation 2004, pp. 51–72 (2004)
9. Seefried, S., Chakravarty, M., Keller, G.: Optimising Embedded DSLs using Template Haskell (2003)
10. Balland, E., Brauner, P., Kopetz, R., Moreau, P.E., Reilles, A.: Tom: Piggybacking Rewriting on Java. In: Baader, F. (ed.) RTA 2007. LNCS, vol. 4533, pp. 36–47. Springer, Heidelberg (2007)
11. Visser, J.: Visitor Combination and Traversal Control. In: OOPSLA 2001: Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, pp. 270–282 (2001)
12. Lämmel, R., Visser, J.: Typed Combinators for Generic Traversal. In: Krishnamurthi, S., Ramakrishnan, C.R. (eds.) PADL 2002. LNCS, vol. 2257, pp. 137–154. Springer, Heidelberg (2002)
13. Lieberherr, K.J.: Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns. PWS Publishing Company (1996)
14. Lieberherr, K., Patt-shamir, B.: Traversals of Object Structures: Specification and Efficient Implementation. ACM Trans. Program. Lang. Syst., 370–412 (1997)
15. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley, Reading (2000)
16. World Wide Web Consontium (W3C): XML Path Language (XPath), Version 2.0, W3C Recommendation (January 2007), http://www.w3.org/TR/xpath20
17. Visser, E., Benaissa, Z., Tolmach, A.: Building Program Optimizers with Rewriting Strategies. In: Proceedings of the International Conference on Functional Programming (ICFP 1998), pp. 13–26. ACM Press, New York (1998)
18. Abrahams, D., Gurtovoy, A.: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series). Addison-Wesley Professional, Reading (2004)
19. 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), pp. 291–310 (2006)
20. Siek, J.G., Lumsdaine, A.: C++ Concept Checking. Dr. Dobb's J. 26(6), 64–70 (2001)
21. Portland Pattern Repository WikiWikiWeb: Hierarchical Visitor Pattern (2005), http://c2.com/cgi/wiki?HierarchicalVisitorPattern
22. Simeoni, F., Lievens, D., Connor, R., Manghi, P.: Language Bindings to XML. IEEE Internet Computing, 19–27 (2003)

23. Magyari, E., Bakay, A., Lang, A., Paka, T., Vizhanyo, A., Agrawal, A., Karsai, G.: UDM: An Infrastructure for Implementing Domain-Specific Modeling Languages. In: The 3rd OOPSLA Workshop on Domain-Specific Modeling (October 2003)
24. Veldhuizen, T.: Expression Templates. C++ Report 7(5), 26–31 (1995)
25. Tambe, S., Gokhale, A.: An Embedded Declarative Language for Hierarchical Object Structure Traversal. In: 2nd International Workshop on Domain-Specific Program Development (DSPD) (October 2008)
26. Becker, P.: Standard for Programming Language C++. Working Draft, N2798=08-0308, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++ (October 2008)
27. Gregor, D.: ConceptGCC: Concept Extensions for C++ (August 2008), `http://www.generic-programming.org/software/ConceptGCC`
28. Lédeczi, Á., Bakay, Á., Maróti, M., Völgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing Domain-Specific Design Environments. Computer 34(11), 44–51 (2001)
29. Balasubramanian, K., Balasubramanian, J., Parsons, J., Gokhale, A., Schmidt, D.C.: A Platform-Independent Component Modeling Language for Distributed Real-Time and Embedded Systems. In: RTAS 2005, pp. 190–199 (2005)
30. Munkby, G., Priesnitz, A., Schupp, S., Zalewski, M.: Scrap++: Scrap Your Boilerplate in C++. In: WGP 2006: Proceedings of the 2006 ACM SIGPLAN workshop on Generic programming, pp. 66–75 (2006)
31. Lämmel, R., Visser, E., Visser, J.: Strategic programming meets adaptive programming. In: Proc. Aspect-Oriented Software Development (AOSD), pp. 168–177 (2003)
32. Alcino, C., Joost, V.: Transformation of Structure-Shy Programs: Applied to XPath Queries and Strategic Functions. In: PEPM 2007: Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, pp. 11–20 (2007)
33. Lämmel, R.: Scrap Your Boilerplate with XPath-like Combinators. In: POPL 2007: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 137–142 (2007)
34. Hejlsberg, A., et al.: Language Integrated Query (LINQ), `http://msdn.microsoft.com/en-us/vbasic/aa904594.aspx`
35. Object Management Group: Unified Modeling Language: OCL version 2.0 Final Adopted Specification. OMG Document ptc/03-10-14 edn. (October 2003)