SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2014; **00**:1–8
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

Software Evolution and the Visitor Pattern: Motivation for a Dynamic Tree Walker

Blair Durkee*, Daniel T. Welch, Murali Sitaraman

School of Computing, Clemson University, South Carolina, SC, 29634, USA

SUMMARY

This paper describes our experiences re-engineering legacy software leading up to, and resulting in the creation of a reusable, reflection-based (dynamic) tree traversal mechanism. The mechanism we describe has come to play a central role in an ongoing software engineering project that that has slowly evolved from a relatively simple language translator, to a sophisticated verifying compiler over a period of nearly fifteen years. The source language we discuss is RESOLVE (Reusable Software Language with Verification): A 'research language' designed for formal specification and verification of object-oriented programs. Being at its core a research language, frequent and continual language-level evolution posed major maintainability challenges to the compiler. Everything from major to minor syntax changes (or additions) would often necessitate compiler wide refactors, spanning the abstract syntax representation, to the logic dictating abstract syntax traversal and all associated subcomponents. Given the development bottleneck posed by this propagation, we decided to build a novel, reflection based, generic walker that utilizes a relatively common pre-post variation of the visitor pattern. We describe this walker, some improvements to its initial design, and conclude with an application, illustrating its important role in code generation for RESOLVE's multiple target languages (C and Java).

Received . . .

KEY WORDS: visitor; design pattern; compiler; language translation

1. INTRODUCTION

Reusability and maintainability are key nonfunctional characteristics of well-engineered software. Where long-running academic research software projects are concerned, these qualities become especially important, but unfortunately, are too often overlooked in favor of rapid fire development – usually culminating with that long sought-after paper or dissertation. This was the very situation in which we found ourselves, with respect to our research compiler and its legacy software components.

Our compiler and its associated tools have been developed over the course of many years and successive generations of students. These tools have evolved and changed significantly, yet many early design decisions still persist. One example is the mechanism used to traverse the abstract syntax tree (AST) that represents parsed code. As with any compiler, this logic is a key component used in multiple stages of compilation such as pre-processing, population, analyzing, semantic checking, translation, etc. The initial implementation of AST traversal worked sufficiently well, but it turned out to be an impediment to the continued evolution of the compiler, thus serving as the motivation for the innovative reengineering solution desribed in this paper. We sought out

*Correspondence to: Blair Durkee, School of Computing, Clemson University, Clemson SC, 29631, USA.

a solution to overcoming our system's highly entrenched, non-reusable, and overall difficult-to-maintain preexisting tree walking strategy – a solution that meets the standards of reusability and maintainability but does not compromise rate of development.

To give some background and context about this component, we will briefly describe the compiler in which it exists – the RESOLVE compiler. RESOLVE is an integrated programming and specification language that seeks to realize the grand challenge of software verification. By writing reusable components that are formally specified, the compiler uses these specifications to produce a number of verification conditions (VCs) for any code programmers might write. These VCs are then sent to an integrated prover which attempts to automatically establish the validity of each VC generated, thus proving the program correct[†]. While this process cannot guarantee software that is specified correctly, it rules out implementation errors by proving programs correct with respect to a given specification.

2. RELATED WORK

Since the emergence of the first compiled, high level languages, ASTs and tree traversal patterns have garnered both an extensive amount of study, and a large (still growing) body of research. While the concept of ASTs and their usage is well established, AST construction, reusability, and long term maintenance remain relevant topics in the software engineering community [cite?]. In this section we consider several existing parsing tools capable of generating ASTs, emphasizing the maintainability of using such tools, and any potential mechanisms they provide for tree traversal.

The first tool that comes to mind is the popular GNU "compiler compiler," Bison[‡] [2]. Provided with a formal grammar, this tool – referred to commonly as a "parser generator" – produces the state machines capable of recognizing a sentence in a given language.

The popular compiler compiler YACC is able to generate a parser which produces a syntax tree. It can generate a tree walker as well, but is limited by a number of factors. The tree walker is generated from a definition file that uses special YACC syntax. The definition file must be updated in tandem with each change to the grammar of the language, and the tree walker code must be regenerated. This puts a strain on maintainability, particularly for developers who might be working on the later stages of the compiler's pipeline such as population or semantic analysis. Small tweaks to the grammar can percolate through the pipeline and break existing code in these later stages.

One example of innovation on the basic model set by YACC is demonstrated by the compiler compiler SableCC [2]. SableCC provides the tools necessary to convert a BNF grammar into Java packages for lexing, parsing, and tree analysis. In addition, it creates a Java class for each node in the AST. The analysis package defines an abstract class which allows for efficient and foolproof implementation of tree traversal logic. The process of parsing the code and walking the tree is all done by generated code, and the developer needs only to implement specific visitor methods to create the analysis and code generation portions of the compiler.

ANTLR (Another Tool for Language Recognition) is another common parsing tool used for tree generation and traversal. The compiler discussed in this paper is based on ANTLR v3, upgraded from previous versions. While ANTLR v4 offers a new approach incorporating sax-dom event style parse tree listeners, we needed a quicker turnaround time than re-writing the grammar and significant portions of the compiler would allow. While there have been many innovations int he field of compiler tree walking, this paper is more specifically about the use of Java reflection to quickly allow more flexibility in accessing an existing tree structure.

[†] *All* VCs must be established to ensure program correctness.
[‡] Bison serves as the modern successor to the original tool, YACC (Yet Another Compiler Compiler)

DOI: 10.1002/spe

```
public abstract
        class ResolveConceptualVisitor {
    visitProcedureDecl(
                ProcedureDecl e) {}
}

public abstract
        class ResolveConceptualElement {
    abstract void accept(
        ResolveConceptualVisitor e);
}

public class ProcedureDecl
        extends ResolveConceptualElement {
    List<Stmt> myStatements;

    public void accept(
                ResolveConceptualVisitor e) {
        v.visitProcedureDecl(this);
    }
}
```

```
public class Analyzer
        extends ResolveConceptualVisitor {

    public void visitProcedureDecl(
                        ProcedureDecl e) {
        table.beginProcedureScope();
        visitStmtList(e.getStatements();
        table.endProcedureScope();
    }

    private void visitStmtList(
                List<Stmt> e) {
        for (Stmt s : e.getStatements()) {
            visitStmt(s);
        }
    }

    public void visitStmt(Stmt e) {
        e.accept(this);
    }
}
```

Figure 1. A flawed implementation of the visitor pattern.

3. INITIAL IMPLEMENTATION AND MOTIVATION

The pipeline for compiling high-level code begins with lexing, parsing, and building an abstract syntax tree. This data structure is a logical representation of the source code, and it will be used in nearly every subsequent stage of the compilation pipeline. The need to traverse the tree in each stage presents a problem of code reuse. The traversal mechanism will be remain the same during each use, but it must invoke different node visitation logic in each compilation stage (e.g., populator, analyzer, code generator). In order to avoid code duplication the traversal must be decoupled from the visitation logic, which is a non-trivial task. This task, however, was simplified by a widely accepted design pattern found in the Gang of Four's seminal work *Design Patterns* [5]. This "visitor pattern" was used in the initial development of the RESOLVE compiler, but it was used imperfectly.

The design of the tree traversal component's first iteration bears close resemblance to the visitor pattern: Each class representing a type of node in the syntax tree contains an `accept` method which can dispatch the appropriate logic through subtype polymorphism. However, the algorithm for traversal was not properly separated from the data structure as the visitor pattern requires.

The very purpose of the visitor pattern is to decouple a data structure from the logic operating on it. While manifestations of this pattern may vary, they must necessarily adhere to that specific design principle of separation. During the initial development of the RESOLVE compiler, a few shortcuts were made to address superficial issues with the visitor implementation. 1 illustrates a snippet of the resulting code.

The traversal logic, according to visitor pattern, should be contained within the `accept` visitor method. In this initial implementation, it has been moved to the `visit` method contained in the `ResolveConceptualVisitor` component. Consequently, every `ResolveConceptualVisitor` will bear the responsibility of traversing the syntax tree. The structure of the data is coupled with the visitation logic in direct opposition to the fundamental principles of the visitor pattern. Thus any change in the tree structure – no matter how minor – will necessitate large, cross-component refactors.

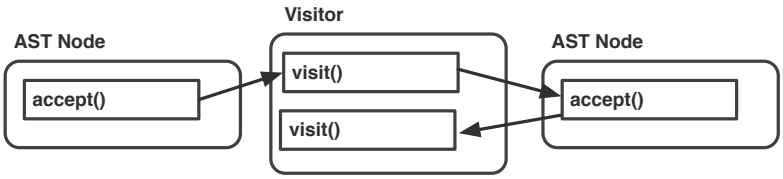

Figure 2. A high level look at the organization of the flawed visitor.

DOI: 10.1002/spe

The reason for the appearance of this critical flaw is not certain to us who inherited this legacy code, but it seems likely to be related to having a singular visit method rather than pre and post visits. Having only one visit method per tree node restricts the traversal to only a pre-ordering or a post-ordering. 1 demonstrates a common scenario for AST traversal–the opening and closing of new scopes in the symbol table–which requires both orderings. The scope is opened as we traverse down the tree and closed as we traverse back up, but of course, this is not possible if we visit the node only once. The legacy code "solves" this problem by placing the traversal logic inside the visitor methods and, in the process, defeats the very purpose of the visitor pattern. It is possible to reclaim a proper visitor pattern implementation by adding preorder and postorder visits. Consider the code with that minor change, shown in 3

```
public abstract
        class ResolveConceptualVisitor {
    preProcedureDecl(ProcedureDecl e) {}
    postProcedureDecl(ProcedureDecl e) {}
}

public abstract
        class ResolveConceptualElement {
    abstract void accept(
        ResolveConceptualVisitor e);
}

public class ProcedureDecl
        extends ResolveConceptualElement {
    List<Stmt> myStatements;

    public void accept(
          ResolveConceptualVisitor e) {
        v.preProcedureDecl(this);
        for (Stmt s : e.getStatements()) {
            s.accept();
        }
        v.postProcedureDecl(this);
    }
}
```

```
public abstract class Stmt
        extends ResolveConceptualVisitor {
    abstract void accept(
        ResolveConceptualVisitor e) {}
}

public class Analyzer
        extends ResolveConceptualElement {

    public void preProcedureDecl(
                        ProcedureDecl e) {
        table.beginProcedureScope();
    }

    public void postProcedureDecl(
                        ProcedureDecl e) {
        table.endProcedureScope();
    }
}
```

Figure 3. A flawed implementation of the visitor pattern.

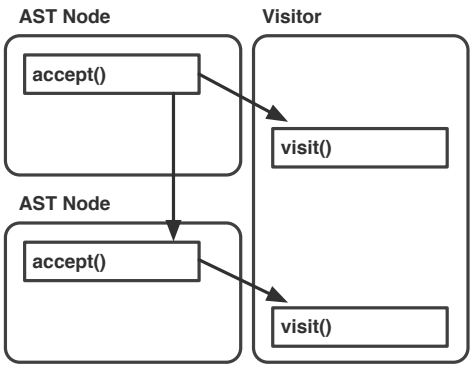

Figure 4. A high level look at the organization of the fixed visitor.

The `ResolveConceptualVisitor` now has fewer methods and the traversal logic is properly decoupled from the visitation logic.

4. A DYNAMIC TREE WALKER

While the corrections demonstrated in 4 could be made, we needed a solution that would not require refactoring of our legacy code. Our existing components needed to continue to work in their present form while we developed new versions of these components. Additionally we believed

DOI: 10.1002/spe

we could develop a solution that would consume much less time, both initially and in the long-term, than a significant refactoring would. Therefore we decided to pursue a third, even more robust implementation that would exist independently of–and work simultaneously with–legacy code.

The solution involves the creation of two new classes: `TreeWalker` and `TreeWalkerVisitor`. `TreeWalker` is the new traversal mechanism which is completely decoupled from both the tree structure and the visitation logic. It dynamically analyzes the tree composition at runtime using Java reflection. This `TreeWalkerVisitor` replaces the old `ResolveConceptualVisitor` class and provides a new abstract visitor class to implement visitor logic for the various components in the compiler. This design does not modify any existing code and still utilizes the `ResolveConceptualElement` AST classes. This allows the legacy code to continue to work alongside the new dynamic traversal component. In other words, old components can continue to work until new components are ready to be dropped in place.

With this new design, `TreeWalker` will dynamically analyze the structure of the tree at runtime and invoke the appropriate visitor methods as it traverses the tree. 5 is a highly simplified version of the traversal algorithm code (see Appendix for complete code).

```
public void visit(ResolveConceptualElement e) {
    invokeVisitorMethods("pre", e);

    for (ResolveConceptualElement node : e.getChildren()) {
        visit(node);
    }
    invokeVisitorMethods("post", e);
}
```

Figure 5. A revised `visit` method.

The `visit` method is a simple recursive, depth-first traversal of the AST. At each level, the procedure will make "pre" call before visiting children and a "post" call after. These calls are made via `invokeVisitorMethods` – a local method used to construct the appropriate visitor method name and dispatch the correct calls using reflection techniques (the details are not relevant to the overall design, but see appendix for this method's code). These calls include pre and post methods for each class (base and derived classes) represented by the AST node object.

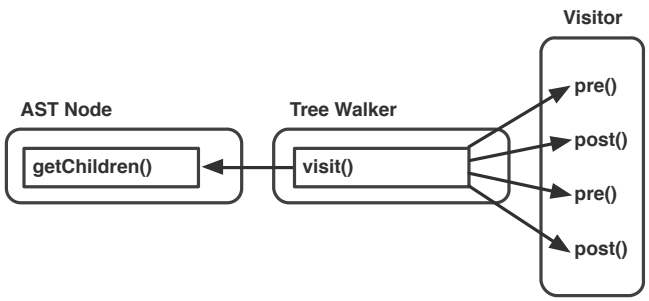

Figure 6. A revised `visit` method.

The logic for retrieving a node's children is contained in the `getChildren` defined in the root AST class hierarchy, using the base `ResolveConceptualElement` class. This design choice allows for more control over the order of traversal. The default base method uses Java reflection to obtain a list of the children for a given node and returns the children in a list of unspecified order. Figure 4 shows a simplified version of this dynamic `getChildren` method (see Appendix for complete code). If the order is important (or needs to be different from the default), then derived classes can override the method with static logic for returning the children.

Use of the dynamic tree walker is simple and straightforward. To traverse an AST and apply appropriate visitation logic, first create an instance of the `TreeWalker` class, passing an instance of a `TreeWalkerVisitor` (such as the populator or code generator) as a parameter to the constructor. Then, simply call `visit` on the root of the abstract syntax tree.

DOI: 10.1002/spe

```
public List<ResolveConceptualElement> getChildren() {
    List<ResolveConceptualElement> children = new LinkedList<>();
    ArrayList<Field> fields = this.getDeclaredFields();

    for (Field field : fields) {
        if (ResolveConceptualElement.class.isAssignableFrom(field.getType())) {
            children.add(ResolveConceptualElement.class.cast(field.get(this)));
        }
    }

    return children;
}
```

Figure 7. An implementation of `getChildren`.

5. RESULTS AND BENEFITS

The dynamic tree walker is a unique and innovative reengineering approach that has yielded a number of benefits for our ongoing research project. The initial benefit was its rapid implementation. The original version of the dynamic tree walker, while admittedly less complex than the iteration seen in the appendix, was implemented by a single developer in the span of a few hours. This allowed us to begin work on new versions of our compiler components with very little delay and with no interruption to its existing operation. In fact, it allowed for simultaneous operation of both new and old components–each sharing the same AST class hierarchy but using their distinct tree traversal mechanisms.

Another major benefit of the dynamic tree walker is that it adds an entirely new layer of abstraction to the visitor pattern. The visitor pattern already mandates the decoupling of the traversal logic from the visitation logic, but our design also separates the traversal logic from the structure of the tree itself. Because the structure of the tree is extracted from the code dynamically at runtime, changes can be made to the AST classes and the tree walker will seamlessly adjust. The visitors, on the other hand, may need to be adjusted in cases where an existing part of the tree was renamed or removed–though perhaps not when adding to the tree. This is due to the fact that visitation logic is, by necessity, coupled with the tree structure.

In many compiler compilers, the relationship between the traversal algorithm and the tree structure is established in a pre-runtime code generation phase. This is usually accomplished by auto-generating AST classes with hard-coded traversal logic extracted from a grammar or other definition file. Our reengineering approach, however, does not require us to rewrite our legacy AST classes (or any part of them). Furthermore, it largely avoids the code generation step. It may be desirable to generate the abstract `TreeWalkerVisitor` class ahead of runtime to simplify the creation of new components by providing method declarations to override (indeed, we have opted for this approach). However, because methods are invoked using dynamically-constructed method names, this is not strictly necessary.

Finally, the dynamic tree walker is has reusable and maintainable. As has been clearly demonstrated, there is very little coupling in our design. This allows the tree walker to be effortlessly reused for any number of components in our compiler. The maintainability is also enhanced by the fact that the traversal logic is contained within a single class rather than distributed over the set of all AST classes. We have already leveraged this benefit by easily making additions and changes to the traversal such as inserting virtual nodes on-the-fly. Overall, the dynamic tree walker has saved a large amount of development time by eliminating the need to rewrite legacy code while also providing a rapidly implementable mechanism for continued iteration and evolution of the software.

6. APPLICATION EXAMPLE: CODE GENERATION

One application of the walking mechanism that has been mentioned in this paper is the code generation phase. Since development began on the RESOLVE compiler, code generation – like the many other phases of compilation – was hindered not only by initial design of the AST visitor, but

DOI: 10.1002/spe

also by the (unusually) steep set of constraints RESOLVE code generation is subject to, including the following.

1. *Correct by construction*: Provided with successfully verified RESOLVE source-code, it falls upon the translator to model, as faithfully as possible, each construct of the source language *within* the target language. This modeling process – which strives to maintain the established correctness of the original source – typically precludes the possibility of any sort of syntax-directed translation – as any code generated fitting such a model inevitably ends up looking wildly different from the original source.
2. *Extensibility*: The design of the translator must allow users to relatively easily tweak the output of a given construct, add support for altogether new constructs (accounting for the rapidly developing nature of the source language), and not preclude the addition of any future target languages. This is ultimately one of the reasons we choose to perform source-to-source translation, as opposed generating byte code such as JVM or LLVM single static assignment form directly: It allows a certain level of flexibility – leaving us free to temporarily sidestep the non-trivial problem of developing (and maintaing) a fully blown byte level interpretation of every RESOLVE construct, in favor of more fruitful, verification related avenues of research.
3. *Reusability*: If two or more supported target languages share similar constructs, the (separate) modules responsible for generating code for each should not duplicate code. Rather, they would ideally be designed to share as much common translation logic as possible, typically via an abstract class or some other means. However, if this is to occur, the translator in question must be designed in such a way that it enforces a strict separation between the logic governing the collection of translation related information, and the actual formatted output of this information.

With these requirements in place, development of a RESOLVE translator hinges on two key pieces: A mechanism for traversing RESOLVE's AST efficiently and automatically (detailed in this paper), and the ability to efficiently separate translation *information collection logic* from translation *output logic*. In this section, we detail – by way of a small example – our approach which utilizes the pre-post methods of the tree walker and the ANTLR tool *StringTemplate* to help us maintain the necessary separation between the model of our translation and view responsible for output.

To illustrate the general process of producing runnable C from RESOLVE, consider the following simple operation:

```
Operation Int_Do_Nothing();
Procedure
        Var X : Integer;
        X := 3;
end Nothing;
```

Shown in 8 is a high level depiction of the steps taken in translating this operation to C. The first box depicts the AST of operation `Int_Do_Nothing`, where nodes are represented as boxes labeled by the constructs they contain. Throughout the walk of the tree, useful information such as the operation's name "Int_Do_Nothing" are extracted from the nodes, and added as parameters to user defined templates[§], in this case: `c_function_def`.

In the context of RESOLVE to C translation, these templates, when filled during the aforementioned pre-post visitor traversal of RESOLVE's AST, help simplify the task of producing arbitrarily complicated, nested blocks of structured C output by keeping translation logic strictly with the C translator, and output logic strictly within the templates.

That is, the only actual work being performed within the C translator is forwarding information collected from individual `ResolveConceptualElement`s, to a series of externally defined templates. This allows us to exploit (in design pattern parlance) a strict model view controller

[§] A template can simply be thought of as a "document with holes" which the user choses when and how to fill.

Prepared using speauth.cls
DOI: 10.1002/spe

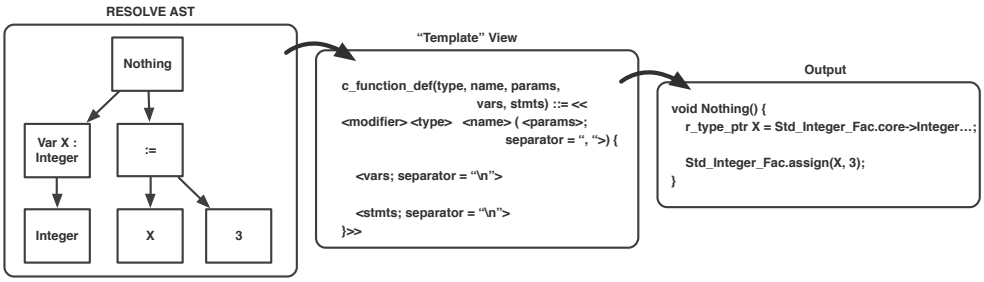

Figure 8. The general flow of information from the AST (first), to user defined templates (middle), ending with formed output (last).

(MVC) separation in the translator's codebase between the mechanism that does the AST visiting (controller), the individual `ResolveConceptualElements` from which we're adding information to templates (model), and the external file containing all available C language templates which shape our output (view) [6].

We feel the approach to tree walking in the paper lends itself well to this strategy of translation, as this separation allows us to easily iterate changes to our generated C (or Java!) code without needing to concern ourselves with the compiler or translator itself.

7. CONCLUSION

8. ACKNOWLEDGEMENTS

REFERENCES

1. Pati T, Hill JH. A survey report of enhancements to the visitor software design pattern. *Software: Practice and Experience* 2014; **44**(6):699–733, doi:10.1002/spe.2167.
2. Levine J, Mason T, Brown D. *Lex & Yacc, 2nd Edition*. Second edn., O'Reilly, 1992.
3. Gagnon E, Hendren L. Sablecc, an object-oriented compiler framework. *Technology of Object-Oriented Languages, 1998. TOOLS 26. Proceedings*, 1998; 140–154, doi:10.1109/TOOLS.1998.711009.
4. Parr T, Fisher K. Ll(*): The foundation of the antlr parser generator. *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, ACM: New York, NY, USA, 2011; 425–436, doi:10.1145/1993498.1993548.
5. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 1995.
6. Krasner GE, Pope ST. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.* Aug 1988; **1**(3):26–49.