SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2014; **00**:1–7
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. As far as long-running academia based 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. Here we describe our solution to overcoming our system's (the RESOLVE compiler's) highly entrenched, non-reusable, and overall difficult-to-maintain preexisting tree walking strategy – a solution which does not compromise rate of development. To understand the context in which this component was designed, and the full implications of its usage, we start with a brief overview of the compiler and its source language.

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 off to an integrated prover which then attempts to automatically establish each VC generated – thus proving the program correct[†]. While

*Correspondence to: Blair Durkee, School of Computing, Clemson University, Clemson SC, 29631, USA.
[†]*All* VCs must be established to ensure program correctness.

this process cannot guarantee software that is specified correctly, it rules out implementation errors by proving programs correct with respect to a given specification.

The compiler, verification, and proof tools have been researched and developed over the course of many years, and successive generations of students. These tools have evolved and changed significantly, though many early design decisions still persist. One example is the mechanism used to traverse the abstract syntax tree (AST) that represents parsed RESOLVE 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 turned out to be an impediment to the continued evolution of the compiler – serving as motivation for the development of a dynamic tree walker to replace this older mechanism.

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[‡] [1]. 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.

3. INITIAL IMPLEMENTATION AND MOTIVATION

The pipeline for compiling high-level code begins with lexing, parsing, and building an AST. 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

[‡]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.

potential problem of code duplication. the traversal method will not change, but separating it from the logic of the populator, analyzer, and other components is a non-trivial task. This task, however, was simplified by a widely accepted design pattern from the gang of four [2]. This "visitor" pattern was used, albeit imperfectly, in the initial development of the RESOLVE compiler.

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 a `accept` method which can dispatch the appropriate logic through polymorphism. However, the algorithm for traversal was not properly separated from the data structure as the visitor pattern requires.

The 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 solve surface issues with the visitor implementation. 1 illustrates a snipped 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. Thus, most changes in the trees structure, no matter how minor, necessitates large, cross component refactors.

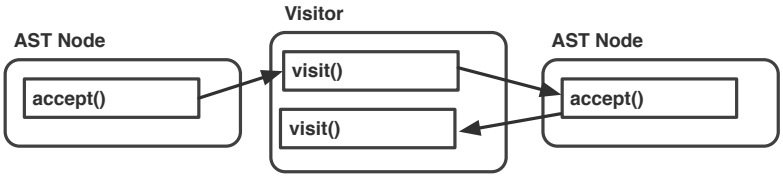

Figure 2. A high level look at the organization of the flawed visitor.

The reason for the appearance of this critical flaw is unknown, but it seems likely to be related to the singular visit method rather than pre-post traversal visits. Consider the same code with that minor change, shown in 3

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

DOI: 10.1002/spe

```
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.

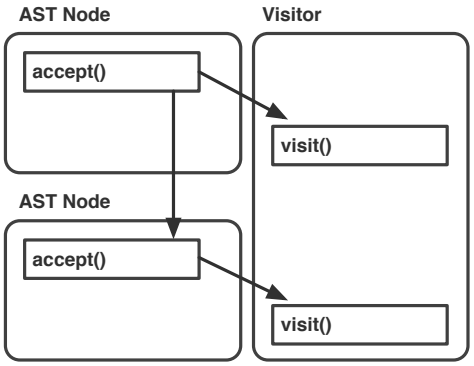

Figure 4. A high level look at the organization of the fixed visitor.

4. A DYNAMIC TREE WALKER

While the corrections demonstrated in 4 could be made, we decided to pursue a third, even more robust implementation that would exist independently of–and work simultaneously with–existing code. We created an entirely new class simply named TreeWalker. This new traversal component is completely decoupled from the data structure and dynamically analyzes the structure at runtime using Java reflection.

Instead of reusing the existing ResolveConceptualVisitor class, we created a new abstract visitor class for the dynamic tree walker–TreeWalkerVisitor. This decision allows the legacy code to continue to work alongside the new dynamic traversal component. One tradeoff is that future visitors have to be entirely rewritten (or at least heavily refactored), but the concern is mitigated, as old components can continue to work until the new components are ready to be dropped in place.

With two new components in place, the Tree Walker can then call the visitor methods as it traverses the tree. 5 is a simplified version of the traversal algorithm.

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 name and dispatch the correct call using reflection techniques.

The logic for retrieving a node's children is contained in the `getChildren` defined in the root RESOLVEs AST hierarchy, the `ResolveConceptualElement` class. This design choice

DOI: 10.1002/spe

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

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

Figure 5. A revised `visit` method.

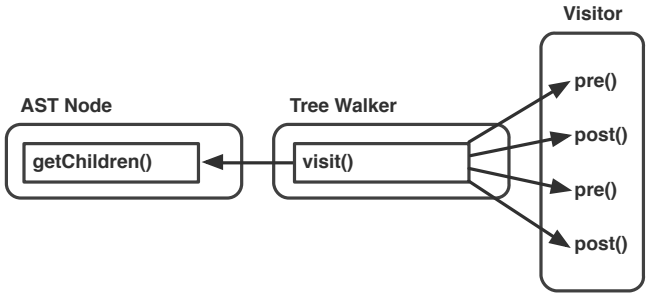

Figure 6. A revised `visit` method.

allows for more control over the method of traversal. The default method uses Java reflection to obtain a list of the children for a given node, and the children are returned in a list of unspecified order. If the order is important (or needs to be different from the default), then inheriting classes can override the method with hard-coded logic for returning the children. Figure 4 shows a simplified version of this method.

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

    //Populate "fields" with the declared fields of the object's defining class

    Iterator<Field> iterFields = fields.iterator();
    while(iterFields.hasNext()) {
        Field curField = iterFields.next();

        Class<?> fieldType = curField.getType();
        if (ResolveConceptualElement.class
                .isAssignableFrom(fieldType)) {
            children.add(ResolveConceptualElement
                .class.cast(curField.get(this)));
        }
    }
}
```

Figure 7. An implementation of `getChildren`.

The visitors we use include a number of typical compiler components and are indicated in 7. Each visitor can override as many or as few visit methods as needed, making the creation of a new visitor quite simple. this has many advantages for maintainability as well. Unlike the original implementation of the visitor pattern, the tree walker allows new methods to be added to a visitor without the need to consider existing code. All the traversal logic is contained within the tree walker class.

5. APPLICATION: CODE GENERATION

One application of the walking mechanism that has been described at length in this paper is its key role in RESOLVE's code generation phase. Since development began on the compiler, code generation – like the many other phases of compilation – was hindered not only by initial design of

DOI: 10.1002/spe

the AST visitor, but 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.

DOI: 10.1002/spe

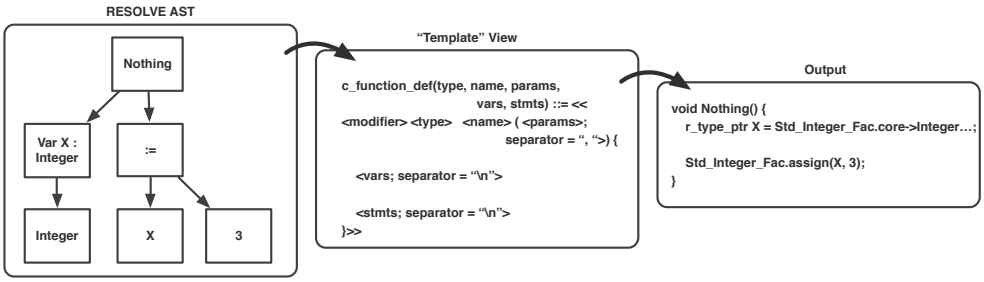

Figure 8. The general flow of information from the AST (first), to user defined templates (middle), ending with formed output (last).

(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) [3].

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.

6. CONCLUSION

7. ACKNOWLEDGEMENTS

REFERENCES

1. Levine J, Mason T, Brown D. *Lex & Yacc, 2nd Edition*. Second edn., O'Reilly, 1992.
2. 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.
3. 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.

DOI: 10.1002/spe