Linking Syntactic and Semantic Models of Java Source Code within a Program Transformation System

Victor L. Winter, Jonathan Guerrero, Alexander James, Carl Reinke

Department of Computer Science
University of Nebraska at Omaha
Omaha, Nebraska
USA
{vwinter, jguerrero, ajames, creinke}@unomaha.edu

Abstract—Static analysis and software manipulation tools are frequently rule-based and draw on a variety of software models in order to achieve their goals. Program transformation languages provide traversal and matching capabilities that are aligned with the core functionality of rule-based systems. Therefore, transformation systems should be considered as candidates for implementing static analysis and manipulation tools. An issue associated with this approach is that transformation systems predominantly operate on syntactic models (abstract/concrete trees) of software. To effectively support in-depth analysis, it is beneficial to integrate the syntactic models used by transformation systems with one or more semantic models.

This paper describes a system, called GPS-Traverse, that establishes a link between syntactic and semantic models of Java software. These models are central to a transformation-based Java source code analysis and manipulation system we are developing called Sextant. Within Sextant, GPS-Traverse provides a coordinate system that is accessible during transformation. These coordinates enable the semantic model to be queried in a context-sensitive fashion during the course of transformation.

Keywords—source code analysis; Java; program transformation; GPS-Traverse; Sextant; TL System;

I. MOTIVATION

Source-code analysis is becoming increasingly important [9]. A Fortify white paper published in 2007 [5], reports that source code analysis is heavily employed across the public and private sectors including: (1) the top five commercial banks, (2) five of the top seven computer software companies, (3) three of the top five aerospace and defense industry leaders, (4) the three largest armed services for the US, (5) three of the leading four accounting firms, and (6) two of the top three insurance companies. In a 2011 white paper from Klocwork [10], Todd Landry argues that source-code analysis can play an important role with respect to software assurance within an Agile development process. A similar assurance-based argument is being made in the medical arena, where the FDA is now recommending the use of static analysis tools for the development of medical device software. In fact, GrammaTech’s CodeSonar is a static analysis tool that the FDA is currently using to investigate failures in recalled medical devices [7]. For a comparison of bug finding tools for Java see [15].

The demand for static analysis (as well as other forms of analysis) continues to grow, especially for high-consequence systems. In particular, the analysis needs articulated in the public, private, and military sectors indicate that critical systems oftentimes have static analysis needs lying beyond the functionality provided by commercial static analysis tools [13]. While it is true that general-purpose tools can be customized to a particular set of interests, in many cases such customization amounts to selecting a subset from a fixed feature set, tuning the values of certain parameters (e.g., to filter false positives), or incorporating additional data from sources external to the tool. Because of these limitations there is a growing need for environments that provide suitable abstractions for creating customized analysis (e.g., domain-specific analysis). Key abstractions of such environments include the ability to:

1) traverse (or otherwise iterate over) a term structure,
2) recognize specific term structures (e.g., via matching),
3) aggregate information,
4) query software models, and
5) transform or manipulate software.

We assert that program transformation systems provide a foundation that is well-suited to the development of such abstractions. Hence, exploring the potential of program transformation frameworks to create environments providing customized analysis capabilities is a worthwhile endeavor.

A. Contribution

This paper describes a system, called GPS-Traverse, that creates a relation between the terms (i.e., portions of source code) visited during a traversal and the contexts in which they structurally reside. This relation provides access to a variety of semantic analysis functions during the course of program transformation (e.g., what is the type of the identifier x with respect to the current context). The result is a transformation system in which semantic information can play an important role in controlling the application of transformations. Such control provides the underpinnings of
a variety of well-known source-code manipulation activities including: refactoring, code migration, slicing, and static weaving (within an aspected-oriented framework).

Based on these ideas we are developing a tool, called Sextant, whose goal is to provide an environment where end-users can specify a wide range of static analysis and manipulation objectives. We plan to achieve this by (1) identifying and developing an appropriate set of semantic analysis primitives, (2) designing a domain-specific language in which analysis functions can be specified, and (3) integrating the semantic primitives and analysis specification language with standard core capabilities provided by a transformation framework. GPS-Traverse is the linchpin of this design.

The remainder of this paper is organized as follows: Section II briefly discusses Sextant and gives an overview of the TL program transformation system and Bascinet, the IDE developed for TL. Section III gives some of the technical details of GPS-Traverse. Section IV outlines how GPS-Traverse can be incorporated into a transformation-based analysis. Section V describes some of the analysis capabilities of GPS-Traverse enhanced transformations. And Section VII concludes.

II. INFRASTRUCTURE

Sextant is a source code analysis and manipulation tool implemented on top of the TL System (a program transformation system). In Sextant, static analysis and manipulation objectives are specified as higher-order transformations. These transformations can be applied to Java source code bases spanning multiple file directories. Traversals such as top-down are used to process syntactic representations of Java source code. During such traversals GPS-Traverse provides critical information. In particular, at any point during a traversal GPS-Traverse can provide answers to a variety of queries such as:

1) What is the canonical name of the context in which the term currently being transformed resides?
2) Is the enclosing context a method, constructor, (anonymous) class, interface, or enum?
3) What local variables are currently within scope?
4) What generic types are currently within scope?
5) What is the canonical name of the context enclosing the context in which the term resides? For example, what is the canonical name of the class containing the method whose body is currently being traversed?

Within Sextant, information provided by GPS-Traverse is used in a variety of ways: to create and update information in database tables (e.g., number of methods in a class) as well as for providing information essential for semantic-based queries (e.g., what context should the semantic model use when resolving an expression that references a given object).

A. The TL System: A Brief Overview

The TL System includes a (1) GLR parser for translating plain text (e.g., ascii representations of programs) into terms, (2) a TL interpreter implemented in SML for rewriting terms, and (3) a powerful pretty-printer that can be used to translate terms into plain text documents.

The terms that TL transforms are parse trees. These terms contain hidden information describing their point of origin. This information includes the name of the file and the row and column number in the file. Point-of-origin information is useful for tracing information within a transformation and can also be used to calculate the number of lines in a (plain text) file.

The TL System is freely available [17] and can be executed from the command line and runs on the Windows, Mac OS, and Unix operating systems.

B. The TL Language

TL [20][21] is a special-purpose language we have developed for expressing transformation-based computation. TL is a partially type-checked language [11] that is tightly integrated with the functional language SML [12]. This design provides a context for expressing computation in a hybrid fashion spanning both transformation-based and function-based programming idioms. For example, the semantic model used by Sextant is implemented in SML.

A TL transformation, can also be called a strategy, and is composed of one or more conditional rewrite rules having the form:

\[ \text{lhs} \rightarrow \text{rhs} \text{ if \{ condition \} } \]

where the conditional portion of a rule is optional. When present, a condition is a boolean-valued expression consisting of match expressions, strategy applications, and SML function calls.

TL supports a labeling mechanism for abstracting strategies as well as standard strategic combinators such as left-biased conditional choice (\(<+\)) and left-to-right sequential composition (\(<\)) for composing strategies. Let \(r_1\) and \(r_2\) denote two strategies and let \(t\) denote a term in our domain of discourse. The expression

\[ (r_1 <+ r_2) \ t \]

denotes the application of the strategy \(r_1 \leftrightarrow r_2\) to the term \(t\). The evaluation of this application proceeds as follows: First, the application \(r_1 \ t\) is attempted. If this application is successful, then \((r_1 <+ r_2) \ t\) reduces to \(r_1 \ t\). If not, the application \((r_1 <+ r_2) \ t\) reduces to \(r_2 \ t\).

In contrast the expression \((r_1 <; r_2) \ t\) always reduces to \(r_1(r_2 \ t)\). In TL, the unsuccessful application of a strategy to a term does not result in failure. In this regard TL is different from standard strategic systems such as Stratego [18] and Elan [4][3]. In TL, the unsuccessful application of
a strategy to a term behaves like an identity (i.e., the term is left unaltered). A similar, though not identical (for technical reasons), behavior can be expressed in Stratego by extending a base strategy with the strategic constant $id$. For example, the result term produced by a Strategy strategy of the form $(s \leftrightarrow id)$ is equivalent to the result term produced by the TL strategy $s$.

TL provides a library of predefined higher-order as well as first-order generic traversals. This library includes standard traversals such as a top-down left-to-right traversal (denoted TDL) and a bottom-up left-to-right traversal (denoted BUL). The expression

$$\text{TDL } s \ t$$

will traverse the term $t$ in a top-down left-to-right fashion applying the strategy $s$ to every term encountered during this traversal.

TL provides a construct called a module for physically partitioning transformation sets and controlling name spaces. In particular, a set of transformations can be grouped into a module, and this module can be stored in a file. Modules can be imported for use by other modules. The contents of a module can be imported using one of two modes opened or closed. When imported in an opened fashion, the transformations are simply copied into the body of the importing module. When imported in a closed fashion, imported transformations must be referenced using a dot-selector. For example, let $s$ denote a strategy belonging to a module $M$. If $M$ is imported in an opened fashion, then $s$ can be referenced directly; otherwise references to $s$ must be of the form $M.s$.

C. Bascinet

Bascinet is a GUI, inspired by the HATS GUI (its predecessor) [19], that is implemented on the NetBeans platform and provides support for the development and execution of TL applications. Bascinet provides syntax directed editors for SML and TL and a number of other features. Conceptually speaking, Bascinet is to TL what Eclipse is to Java.

Bascinet and the TL System are integrated in a manner that seamlessly supports the application of transformations (i.e., TL programs) to file hierarchies. A developer simply selects the transformation they want to apply together with the file or file hierarchy to which the selected transformation should be applied.

Bascinet supports two distinct application modes: (1) a discrete mode application in which the selected transformation is applied in a repetitive manner to each file in a file hierarchy, and (2) a continuous mode application in which the selected transformation is applied – a single time – to the entire contents of a file hierarchy.

Continuous mode application is very useful when the entity to be transformed has been distributed across a number of files. For example, using this application mode it is straightforward to develop transformations for gathering a wide variety of metrics over a Java code base that spans many files over a number of folders.

Bascinet allows the developer to control to which file extensions (e.g., dot-java) a transformation should be applied. From a practical standpoint, this is an important feature when applying a transformation to a folder hierarchy consisting of hundreds of folders and thousands of files. For example, Java libraries occasionally contain files having extensions other than the dot-java extension. Applying a Java-oriented transformation to a non-Java file will result in failure. The ability to exercise control, based on file-extensions, during the application of a transformation permits transformations to be applied directly to Eclipse workspaces.

III. GPS-TRAVERSE: TECHNICAL DETAILS

The Java Language Specification defines a declared entity as follows:

A declared entity is a package, class type, interface type, member (class, interface, field, or method) of a reference type, parameter (to a method, constructor, or exception handler), or local variable.

GPS-Traversal defines a context as a construct that delineates the scope of (1) a declared entity, or (2) a generic type. In most cases, a context can be easily identified within Java source code using standard first-order matching. However, to accomplish this with a minimal “matching footprint” requires some modifications to the Java grammar.

A. Grammar Engineering

A primary goal of GPS-Traversal is to provide near-transparent support for tracking the entry and exit of contexts within a standard generic top-down left-to-right (i.e., TDL) traversal. To accomplish this in the most direct manner possible, we instrument the Java grammar with additional nonterminals to expose context relevant information to the first-order matching used by the transformation system. An example of such grammar instrumentation for class declarations is shown in Table I. Note that in a TDL traversal the last nonterminal seen prior to exiting a class will be <ClassEnd>. The grammar used by Sextant has instrumented productions exposing entry/exit points for (1) classes, (2) enums, (3) interfaces, (4) annotations, (5) anonymous classes, (6) methods, (7) constructors, (8) blocks, and (9) for-loops (including for-each loops).

In most cases, context entry is conceptually clear. For example, if a TDL traversal encounters either the nonterminal <NormalClassDecl> or <ClassId>, then the traversal

1Other tracking options are also possible, but lie beyond the scope of this paper.
has entered a context corresponding to a class declaration. However, cases do exist involving local variables where the context entry point is not as distinct. Consider the code fragment shown in Figure 1. In the (inner) block occurring within the body of the method \( f() \), the first assignment statement references the field \( j \), while the second assignment statement references the local variable \( j \).

```java
class A {
    int j = 1;
    void f() {
        j = j + 1; // field
        int j = 0;
        j = j + 1; // local
    }
}
```

Figure 1. Contexts capturing local variables.

### B. Functional Components: Context Stacks

GSP-Traverse stores context information in global data structures, implemented in SML, that lie outside the standard transformational framework. The information in these structures persists beyond the application of individual transformations and can be directly accessed from within the conditional portion of any transformation. Suitable abstractions for accessing and querying context information is under development. In particular, we are looking at the functional needs associated with the implementation of a variety of rule-based analysis systems such as Hammurapi, PMD, and Findbugs. We are also considering functional requirements of refactoring and static weaving. To date, we have implemented a library of SML functions that includes the following:

- **getContextKey** – this function returns a key corresponding to the canonical name of the current context. This key is needed to appropriately query the semantic model of the subject Java code base. It should be noted that not all contexts (as we have defined them) have keys. In such cases, `getContextKey` returns the most specific key possible. For example, if a subject term lies within a (nested) block occurring within a method (see Figure 1), then the key returned will be the canonical name for the method.

- **getEnclosingContextKey** – This returns the canonical name of the context (e.g., class) in which the most specific key (e.g., method) resides.

- **contextKind** – This predicate takes as input a string denoting a context kind and returns `true` if the kind of the current context matches the input kind; otherwise `false` is returned. Kinds of contexts only include those for which keys exist. These are: `classes`, `enums`, `interfaces`, `annotations`, `anonymous classes`, `methods`, and `constructors`.

- **enclosingContextKind** – This predicate takes as input a context kind and compares this string to the kind of the context (e.g., class) enclosing the current context (e.g., method).

- **isVar** – When passed an identifier \( x \), this SML function returns `true` if \( x \) is a local variable and `false` otherwise.

- **inFrameConstruct** – This predicate takes as input a identifier \( x \) and a string \( s \) describing a structure in which a variable may be defined (i.e., method, constructor, block, and for-loop). If the current \( x \) is declared in the structure \( s \), then `inFrameConstruct` returns `true`; otherwise `false` is returned.

- **isGeneric** – When passed a type identifier \( x \), this SML function returns `true` if \( x \) is a generic type and `false` otherwise.

### C. Transformational Components: Context, Local, and Generic

GSP-Traverse consists of three transformation modules: (1) Context, (2) Local, and (3) Generic. Highlights of the Context module are shown in Figure 2. In particular, there is an enter/exit strategy (aka transformation) for each context. The body of the `addClass` strategy is shown in full and consists of a rule whose conditional portion (1) checks that the parse tree to which it is being applied is a class, and (2) calls the SML function `GPS_enterContext` updating the global data structure for contexts with the appropriate information (see Section III-B). Note that the `exitClass`
strategy is trivial and simply makes a call to the SML function GPS_exitContext when it is applied to a parse tree whose root is the nonterminal symbol <ClassEnd>.

The Local and Generic modules are conceptually similar to the Context module. The Local module contains strategies for tracking (1) formal parameters of methods and constructors, (2) variables declared within blocks, and (3) variables declared within for-loops (including for-each loops). Collectively, we refer to these declarations as local variable declarations. It should be noted that additional grammar engineering is needed to track local variables. The most interesting such modification involves for-loops and is shown in Table II.

IV. USING GPS-TRaverse

Figure 3 shows a fragment of a TL module we have implemented that determines the cyclomatic complexity (CC) of methods and constructors over a targeted source code base. The module shown imports the GPS.Locator module which provides access to all the GPS-Traversal transformations. More specifically, the GPS.Locator exports two transformations, enter/exit, that are respectively used to track (1) the entrance/exit of Java contexts, (2) local variables, and (3) generic types. These transformations can be integrated with the actual CC analysis transformation by (1) creating a strategy consisting of three sequentially composed sub-strategies, and (2) applying the resulting strategy to compilation units using a generic traversal such as a top-down traversal. As was mentioned in an earlier footnote, GPS-Traversal can be utilized in other ways, but the discussion of this lies beyond the scope of this paper.

In general, importing GPS.Locator (in an opened fashion) to perform a “desired-analysis” results in a strategy having the following form:

TDL (enter <; desired-analysis <; exit)

After this top-level strategy has been created, transformations for the desired-analysis can be written independently of GPS-Traversal and queries to the information collected by GPS-Traversal can be made at any time. It is worth mentioning that this approach makes some assumptions about how additional term traversals are used within the analysis transformations. In particular, nested/local traversals should either (1) not be used to perform analysis requiring GPS-Traversal information, or (2) actively manage GPS-Traversal when this information is needed. In practice, we have not found these constraints to be particularly limiting.

V. APPLICATION

Currently, Sextant provides a transformational setting in which it is relatively straightforward to gather a range of metrics and perform a variety of semantic analysis over large code bases (e.g., Sextant can easily handle code bases consisting of 250K LOC). Examples of metrics and analysis currently supported include:

- The total source lines of code.
- The total number of classes, fields, methods, and constructors within a code base.
- The number of methods (NOM) within a class as well as their mean ($\mu$) and standard deviation ($\sigma$) across all classes within a code base.
- The cyclomatic complexity (CC) of methods and constructors as well as their mean ($\mu$) and standard deviation ($\sigma$).
- The total number of initialization blocks and their locations within the source code.
- The total number of anonymous classes.
- The total number of class declarations occurring within a method or constructor. For example, the subset of the Java Standard Edition (SE) base libraries comprised of {java.io, java.lang, java.math, java.nio, java.util} contains 257,163 lines of code, has no class declarations within constructors, and contains only one class declaration within a method.

One of our current goals is to validate (and refine) the framework of Sextant by implementing the rule-sets of a variety of static analysis systems such as Hammurapi, PMD, and FindBugs. Each of these tools document the rule-sets they implement. Within these static analysis systems, rules can be classified as being syntax-based or semantic-based. Examples of syntax-based and semantic-based rules are shown below.

Syntax Rule 1: Hammurapi-002: Empty catch blocks are prohibited.

Semantic Rule 1: Hammurapi-086: Use equals() instead of == or != to compare objects.
A pure transformational setting is well-suited to determining source code conformance to syntax-based rules. Determining conformance to semantic-based rules employs one (or more) semantic models of the software. In this case, GPS-Traverse is needed to link transformation-based actions performed on the syntactic model of the software to a corresponding semantic model of that software.

A. Constructing a Call-Graph

GPS-Traverse and the link it provides to a (preconstructed) semantic model can be used for a variety of purposes – other than just the enforcement of semantic-based rules. In this section, an example is presented of the role played by GPS-Traverse in the construction of a strategy that produces the call-graph for all the methods of a code base. Specifically, Figure 2 shows a Sextant transformation that prints the call-graph of a code base to the standard output. This transformation uses GPS-Traverse information (shown highlighted) in four places: (1) to determine whether the term being visited occurs within a method, (2) to obtain the canonical name of the method in which a call occurs, (3) as a parameter to the semantic query resolve, and (4) as the source node of a call-graph edge. The semantic function sml.resolve determines the canonical name of the method call – an analysis which takes into account a number of factors.
of things including overloading, conversions, subtyping, and visibility.

VI. RELATED WORK

PMD [13] is a tool that performs static analysis on Java source code. Writing a Java source code based rule in PMD involves: (1) manual construction of an AST for the desired source code using a JavaCC generated parser, (2) creation of a visitor pattern for the (root note of the) targeted AST pattern, (3) addition of the newly created rule to an XML ruleset, and (4) implementation (in Java) of a pattern matcher for recognizing the ASTs matched by the visitor pattern. Note that when compared to other tools, rule creation for PMD is considered to be easy. In a transformational framework, step 1 is unnecessary, step 2 is unnecessary if the transformation (aka rule) is embedded within a traversal, step 3 requires simply naming the new rule (no XML is needed), and step 4 can be accomplished at a very high level of abstraction since first-order matching is a primitive operation in transformation systems. Our observation is that the complexity of PMD rule creation is driven largely by the abstractions provided by the Java compiler, and the underlying software model (an object-oriented representation of an AST) which triggers the need for visitor patterns and explicit low-level pattern matching. Creating new rules is therefore nontrivial and requires an in depth understanding of not only the static analysis technique, but also the tool (PMD) and details of the language and its compiler.

DMS [2] is a commercial software reengineering toolkit supporting program transformation as well as other forms of analysis and manipulation. A primary design objective of DMS is to create an infrastructure supporting very large (50 million SLOC) industrial applications. DMS provides analysis and manipulation capabilities for a number of languages including Java. Within a Java domain, a number of semantic models of software are created. These models include: symbol tables, reaching definitions, and call graphs. Semantic models can be accessed via procedural programming and some capability is also provided for accessing semantic models via arbitrary predicates during transformation.

JTransformer [16,9] is a Java source-code analysis and manipulation system developed within a logic programming framework. JTransformer models Java source code in terms of an abstract syntax tree (AST). This AST is internally represented as a Prolog database. As a result, Prolog’s abstractions (e.g., unification and backtracking) can be leveraged to construct queries of the AST. These queries can be used to drive conditional transformations of the logic fact base. Transformation of the source code is accomplished by propagating changes to the AST back to the source code. Conceptually, Sextant and JTransformer share a number of similarities. A primary difference is the language for specifying analysis and manipulation (what is sometimes referred to as the query language). JTransformer queries operate on the AST and leverages the implicit backtracking capability of Prolog. JTransformer’s AST is a rich model of the software and is similar to the semantic model we implement in SML. Sextant queries operate directly on source code and leverage the traversal capabilities of the TL System – creating a layer of abstraction between the semantic model and the constructs upon which queries are based. Sextant’s transformational origins also provide explicit control over traversal. For a comparative study of code query technologies see [1].

VII. SUMMARY AND CONCLUSION

The importance of source code analysis continues to grow. The analysis needs of safety-critical systems oftentimes lies beyond commercial static analysis tools. This has motivated interest in the development of tools providing suitable abstractions for the construction of domain-specific analysis.

Program transformation represents a mature discipline providing a number of primitive operations (e.g., traversal and first-order matching) central to source code analysis and manipulation. As such, transformation technology offers a suitable foundation for tools capable of domain-specific analysis and manipulation.

The inclusion of semantic models into a syntax-based transformational framework results in a significant enhancement of the capabilities of the overall system. In order to take full advantage of this composition, the syntactic model operated on by the transformational framework must be linked to the semantic model. GPS-Traverse is a collection of transformations and sml functions that provides this link.

ACKNOWLEDGMENT

This work was in part supported by the United States Department of Energy under Contract DE-AC04-94AL85000. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy.

REFERENCES


strategy CallGraph:
  <SelectorOptExpression>_methodCall -> <SelectorOptExpression>_methodCall
  if {
    isMethodCall <SelectorOptExpression>_methodCall
    andalso sml.GPS_contextKind("method")
    andalso <key>_methodContext = sml.GPS_getContextKey()
    // semantic query
    andalso <key>_calledMethod = sml.resolve(<key>_methodContext,<SelectorOptExpression>_methodCall)
    andalso sml.outputPP(<key>_methodContext)
    andalso sml.output(" calls ")
    andalso sml.outputPP(<key>_calledMethod)
  }
strategy isMethodCall:
  // basic call
  SelectorOptExpression[ ] <TypeArgsOpt>_1 <Id>_1 <Arguments>_1 ->
  SelectorOptExpression[ ] <TypeArgsOpt>_1 <Id>_1 <Arguments>_1 []
  <+ // embedded call
  ...

Figure 4. A Sextant rule for constructing the call graph of all methods within a code base.


[13] PMD. http://pmd.sourceforge.net/pmd-5.0.0/


