Minimally Invasive Interpreter Construction

– How to reuse a compiler to build an interpreter –

Christoph Schinko
Institut für ComputerGraphik
und WissensVisualisierung (CGV)
Technische Universität Graz, Austria

c.schinko@cgv.tugraz.at

Torsten Ullrich[1], Dieter W. Fellner[1,2]
[1] *Fraunhofer Austria, Graz, Austria*
[2] *Fraunhofer IGD & TU Darmstadt, Germany*

torsten.ullrich@fraunhofer.at
d.fellner@igd.fraunhofer.de

***Abstract*—Scripting languages are easy to use and very popular in various contexts. Their simplicity reduces a user's threshold of inhibitions to start programming – especially, if the user is not a computer science expert. As a consequence, our generative modeling framework *Euclides* for non-expert users is based on a JavaScript dialect. It consists of a JavaScript compiler including a front-end (lexer, parser, etc.) and back-ends for several platforms. In order to reduce our users' development times and for fast feedback, we integrated an interactive interpreter based on the already existing compiler. Instead of writing large proportions of new code, whose behavior has to be consistent with the already existing compiler, we used a minimally invasive solution, which allows us to reuse most parts of the compiler's front- and back-end.**

Keywords-JavaScript; generative modeling; procedural modeling; compiler; interpreter

I. Introduction

As John Ousterhout has written in *Scripting: Higher Level Programming for the 21st Century* [1], "Scripting languages such as Perl and Tcl represent a very different style of programming than system programming languages such as C or Java. Scripting languages are designed for 'gluing' applications; they use typeless approaches to achieve a higher level of programming and more rapid application development than system programming languages. Increases in computer speed and changes in the application mix are making scripting languages more and more important for applications of the future."

Therefore, scripting languages are not only a common way to automate repeated tasks, but also a relevant tool in algorithm design – gluing existing algorithms and data structures to new solutions.

As pointed out by Ousterhout [1] conventional system programming languages are too 'rigid' for many tasks in contrast to scripting languages, whose flexibility has to be paid by performance.

In order to trade off both, we combined ahead-of-time compilation techniques with just-in-time compilation methods to an interactive interpreter. The result is in interactive environment, in which algorithms can be designed, tested, etc., and whose consistent data structures can be exported and compiled to an application at any time. In this way, we combine the advantages of both worlds.

The field of application as well as the context of this work is presented in Section "II. Related Work". A description of the used compiler has already been published [2], [3] and is summarized in Section "III. Compiler Construction". Based on this compiler, Section "IV. The Interpreter as a Retrofitted Compiler" illustrates the needed extensions to implement an interpreter.

II. Related Work

Originally, scripting languages like JavaScript were designed for a special purpose, e.g., to be used for client-side scripting in a web browser. Nowadays, the applications of scripting languages are manifold. JavaScript, for example, is used to animate 2D and 3D graphics in VRML [4] and X3D [5] files. It checks user forms in PDF files [6], controls game engines [7], configures applications, and performs many more tasks.

A. Field of Application

Scripting geometric objects – also known as generative and procedural modeling – has gained attention within the last few years [8]. The main advantage of generative modeling techniques is the included expert knowledge within an object description. For example, classification schemes used in architecture, archaeology, civil engineering, etc. can be mapped to procedures. In combination with documentation and annotation techniques established in software engineering, 3D objects are easily identifiable by digital library services (indexing, markup and retrieval) on a textual basis.

From a historical point of view, the first procedural modeling systems were Lindenmayer systems [9], or L-systems for short. These early systems, based on grammars, provided the means for modeling plants. The idea behind it is to start with simple strings and create more complex strings by using a set of string rewriting rules.

Later on, L-systems are used in combination with shape grammars to model cities [10]. Parish and Müller presented a system that generates a street map including geometry for buildings given a number of image maps as input. The

 ISBN: 978-1-61208-222-6

resulting framework is known as *CityEngine* – a modeling environment for *CGA Shape*.

Havemann takes a different approach to generative modeling. He proposes a stack based language called *Generative Modeling Language* (GML) [11]. The postfix notation of the language is very similar to that of *Adobe Postscript*.

B. Programming Languages and Paradigms

Generative modeling inherits methodologies of 3D modeling and programming, which leads to drawbacks in usability and productivity. The need to learn and use a programming language is a significant inhibition threshold especially for archaeologists, cultural heritage experts, etc., who are seldom experts in computer science and programming. The choice of the scripting language has a huge influence on how easy it is to get along with procedural modeling.

Processing is a good example of how an interactive, easy to use, yet powerful, development environment can open up new user groups. It has been initially created to serve as a software sketchbook and to teach students fundamentals of computer programming. It quickly developed into a tool that is used for creating visual arts [12]. Processing is basically a Java-like interpreter offering new graphics and utility functions together with some usability simplifications.

Offering an easy access to programming languages that are difficult to approach directly reduces the inhibition threshold dramatically. Especially in non-computer science contexts, easy-to-use scripting languages are more preferable than complex programming paradigms that need profound knowledge of computer science. This is why we use JavaScript – a beginner friendly, structured language.

The success of Processing is based on two factors: the simplicity of the programming language on the one hand and the interactive experience on the other hand. The instant feedback of scripting environments allow the user to program via "trial and error". In order to offer our users this kind of experience, we enhanced our already existing compiler to an interactive environment for rapid application development.

C. Euclides – a JavaScript platform for Cultural Heritage

In the context of Cultural Heritage, the Generative-Modeling-Language (GML) is an established procedural modeling environment designed for expert users [13]. The aim of the *Euclides* modeling framework [14] is to offer an easy-to-use approach to facilitate these platforms. The translation mechanism for GML within *Euclides* has already been described in "Euclides – A JavaScript to PostScript Translator" and presented at the International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking [2].

III. Compiler Construction

This section focuses on the existing compilation pipeline of the Euclides framework. The framework consists of several stages to translate JavaScript code to a number of target languages. Most parts are implemented in Java apart from the parser which is generated using a third-party tool (see Figure 1).

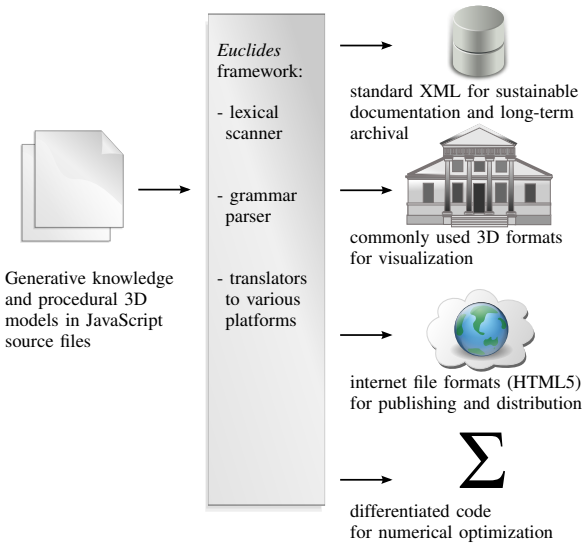

Figure 1. The meta-modeler approach of the *Euclides* framework has many advantages. In contrast to script-based interpreters, *Euclides* parses and analyzes the input source files, builds up an abstract syntax tree (AST), and translates it to the desired platform. Its platform and target independence as well as various exporters for different purposes are the main characteristics of *Euclides*. This innovative meta-modeler concept allows a user to export generative models to other platforms without losing its main feature – the procedural paradigm.

An editor component feeds the first stage of the framework: lexer and parser. For semantic recognition of the input source code, JavaScript syntax needs to be analyzed. All rules, which define valid JavaScript code, form its grammar. For each language construct available in JavaScript, this set of rules is validating syntactic correctness. At the same time actions within these rules create the intermediate structure that represents the input source code – a so-called abstract syntax tree (AST).

The resulting AST is the main data structure for the next stage: semantic analysis. Once all statements and expressions of the input source code are collected in the AST, a tree walker analyzes their semantic relationships, i.e., errors and warnings are generated, for instance, when they are used but not defined, or defined but not used.

Having performed all compile-time checks, a translator uses the AST to generate platform-specific files; e.g., java source code for the JVM platform. In other words, this task involves complete and accurate mapping of JavaScript code to constructs of the target language. A translation in the target language needs to be available for each statement or expression found in the AST. Usually, a direct mapping to data types or operators in the target language is not possible.

 ISBN: 978-1-61208-222-6

Therefore, auxiliary methods and data structures within the target language are needed to mimic JavaScript behavior.

A. Parser

The parser for JavaScript is written using ANother Tool for Language Recognition (ANTLR) [15]. ANTLR provides a framework for constructing recognizers, interpreters, compilers, and translators from grammatical descriptions. It relies on a strategy called *LL(*)* parsing, which extends the *LL(k)* parsing strategy with lookahead of arbitrary length. Using this framework, lexer and parser are generated to syntactically check the provided input for JavaScript compliance.

```
public interface ASTFactory {

    public static interface Tree {
       // tree traversal methods; e.g.
       public Tree getUp();
       public Tree[] getDown();
    }

    public static interface
       Expression extends Tree {
       // validation
       public void validate(
          ErrorHandler errorHandler);
    }

    public static interface
       Statement extends Tree {
       // validation
       public void validate(
          ErrorHandler errorHandler);
       // original source code ref.
       public int getLine();
       public String getFileName();
    }

    public static interface
       TryCatchBlock {
       // This pure markup interface
       // is used to ensure type
       // compatibility.
    }

    // the factory methods; e.g.
    public Statement statementTry(
       String filename, int line,
       Scope scope, Statement statement,
       TryCatchBlock catchBlock,
       TryFinallyBlock finallyBlock);

    // the factory utility methods
    // to create optional terms; e.g.
    public TryCatchBlock utilTryCatchBlock(
       Expression identifier,
       Statement statement);
}
```

Source code 1. This source code excerpt shows the main components of the abstract AST factory used by the Euclides parser to build up an abstract syntax tree.

A first step is to convert a sequence of characters into a sequence of tokens, which is done by special grammar rules forming the lexical analysis. For instance, only a limited number of characters is allowed for an identifier: all characters A-Z, a-z, digits and the underscore are allowed with the condition that an identifier must not begin with a digit or an underscore. These lexer rules are embedded in another set of rules – the parser rules. They are analyzing the resulting sequence of tokens to determine their grammatical structure. The complete grammar consists of a hierarchical structure of rules for analyzing all possible statements and expressions that can be formed in JavaScript, thus forming the syntactic analysis. Rules can be enriched with so-called actions. These actions create the intermediate AST structure.

Within these actions, an abstract factory, like described in [16], called `ASTFactory` is used to create necessary instances of statements and expressions for the AST. An excerpt of the abstract factory including selected inner interfaces is listed in Source Code 1.

The statements and expressions mentioned in the `ASTFactory` are defined as static, inner interfaces `Statement` and `Expression` within the definition of the factory. Both interfaces extend a common interface called `Tree`. The use of a factory has the advantage to be able to replace their implementations without touching the grammar. Additionally, markup interfaces are used to ensure type compatibility, because during AST construction, sub-parts of the AST are created bottom-up via utility methods. These parts are collected and passed to the corresponding parent rule. For example, the AST of the listing in Source Code 2 is created via the following factory calls.

```
try {
   doSomething();
} catch (exception) {
   repairSomething();
   print("caught exception " + exception);
}
```

Source code 2. The catch-block of a JavaScript try-statement automatically declares and defines a variable. In this example it is called `exception`.

The optional catch-block is parsed by a sub-rule with actions, which call the factory method `utilTryCatchBlock`. This method returns an instance of the markup interface `TryCatchBlock`, which can only be passed to a `statementTry` method. This method itself is called in the corresponding rule to match a try-statement. In this way, complex grammar rules are split up into several simpler rules while using the abstract factory pattern and maintaining type safety.

The signature of the `statementTry` call reveals some properties that are passed to the factory by all statements: the source code's file name and line together with the current scope. In case of `statementTry`, the statement to try, the optional catch-block as well as the optional finally-block are also passed to the factory. (Please note, at least one optional block must be non-null.)

 ISBN: 978-1-61208-222-6

B. Abstract Syntax Tree

In JavaScript, the top-level rule of an AST is always a simple list of statements – no enclosing class structures, no package declaration, no inclusion instructions, etc. Each statement contains all included substatements and expressions as well as associated comments. Furthermore, our AST stores additional formatting information (number of new lines, white spaces, tabs, etc.), which offer the possibility to regenerate the original input source code just using the AST.

During the validation step, this tree structure is extended by reference and occurrence links; e.g., each method call references the method's definition and each variable definition links to all its occurrences.

```
@Override
public void forRangeNoArray( String filename, int line) {
   warning(filename, line,
      "The range expression of this for-statement is not"
      + " an array. It will be casted automatically, "
      + "which might lead to undesired results.");
}
```

Source code 3. The Euclides compiler includes a simple and limited type inference implementation. Its main purpose is to recognize common pitfalls of JavaScript source code and to present reasonable warnings.

Having assured that all compile-time checks are carried out, symbols are stored in a so called namespace. During validation, this data structure is used to detect name collisions (e.g. redefinition of variables) and undefined references (e.g. usage of undeclared variables). In addition, a simple type inference system tries to determine the variables' types. As this system is incomplete, it cannot be used for compile time optimizations (e.g. mapping to native data types), but it can be used for warnings and recommendations. To provide meaningful error messages is an important aspect with regard to language processing. In Euclides, an error handler is responsible for collecting and preparing error and warning messages. This functionality is not only used during AST construction to deal with syntactic issues, but also for semantic validation as well. A total of 52 different errors and warnings can be issued. For example, if the type inference system checks the range expression of a for-in loop, it expects an array. If it finds a different type, the warning routine listed in Source Code 3 is issued.

C. Translator to Java – the Compiler Backend

The translation backend for the target language Java is not as straightforward as the similarity in names between Java and JavaScript would suggest. Although they have some similarities, the concepts of both languages show major differences. Java is a statically typed, class-based, general-purpose programming language.

Because of the conceptual differences in the typing system, it is not only unpractical, but impossible to project all JavaScript data types onto built-in Java data types. In JavaScript, there is no difference between integer numbers and floating point numbers. Just one data type called `Number` holds any type of number. Other differences can be found when comparing the remaining data types. Also dynamic typing is not a language feature of Java – as a consequence, each JavaScript data type is re-built in Java to match its functionality.

A total of seven data types are implemented in classes having a common interface called `Var`. These data types are: `VarUndefined`, `VarBoolean`, `VarNumber`, `VarString`, `VarArray`, `VarObject`, and `VarFunction`. A number of access functions and conversion methods are available for all data types. All internal functions provide an additional parameter that always refers to a table entry, which references the corresponding JavaScript file and line number. In this way, warnings can be generated at runtime, if implicit conversion takes place. For example, the implementation of an array access includes the statement `Log.variableTypeChangeImplicit(ii);`. In the messages table (generated by the compiler) there is an entry `#ii` that provides reasonable information needed for a runtime warning.

The access functions reveal the implementation details and the internal Java data types used:

- **Boolean**: The mapped Java data type is `boolean`.
- **Number**: A JavaScript number is mapped to `double`.
- **String**: String is mapped to `String`.
- **Array**: A JavaScript array is realized using the collection `java.util.ArrayList<Var>`.
- **Object**: And an object in JavaScript is mapped to `java.util.HashMap<String, Var>`.
- **Function**: A JavaScript functor is realized in Java as a function pointer using abstract objects.

The instantiation of variables within the generated Java code is performed using factory methods like `Factory.initString(String text)`. Furthermore, all JavaScript operators need to be recreated in Java as well.

A total of 49 operators grouped in unary, binary and tertiary operators are available. Each operator is applied via a method call and can therefore be exchanged easily. These concepts are demonstrated in Source Code 4, which shows the implementation of the binary subtraction operator found in JavaScript. In case at least one of the operands is not of type number, a warning is generated. The operator returns a new number initialized with the result of the subtraction operation of the internal Java data types used.

 ISBN: 978-1-61208-222-6

```
public static Var SUB(int ii, Var v1, Var v2) {
   if (!v1.getType().equals(Type.NUMBER)
    || !v2.getType().equals(Type.NUMBER))
     Log.deviantOperatorCallNoNumber(ii) ;
   return Factory.initNumber(
      v1.toNumber() - v2.toNumber());
}
```

Source code 4. During the translation of JavaScript to Java, all JS-operators are mapped to corresponding Java-based static method calls, which implement their behaviour.

The factory pattern has been chosen for the generated Java code in order to easily replace the mapping of JS variables and operators to different implementations. In this way, we realized a compiler with included, automatic derivation; i.e. within the generated code we can evaluate both: a function $f(x_1, \ldots, x_k)$ as well as its partial derivatives $\frac{df}{dx_i}$. This technique offers the possibility to use standard optimization algorithms to solve numerical optimization problems [17].

All variables defined in the JavaScript source code are collected in the namespace. The Java translator backend, however, distinguishes between variables defined in global scope and local variables. A single class called `Variable` is created holding global variables as static objects. All other variables, e.g. those defined in a function, are exported in-place. Functions itself are mapped to Java functions and are collected in a class called `Function`.

All expressions are exported in their respective embedding statement. A distinction between global and local scope is made in case of the statements. All locally defined statements, e.g., statements defined within a function, are exported in-place. Global statements are collected in a class called `Main` and are executed from the Java main method.

IV. The Interpreter as a Retrofitted Compiler

As stated before, the simplicity of a programming language is only one factor of a successful development environment. Reasonable feedback and an interactive experience are also important. In order to offer our users this kind of experience, we enhanced our already existing compiler to an interpreter. A similar approach to combine interpretation and compilation has been presented by Anton Ertl and David Gregg [18], but in contrast to our system, they start with an interpreter and end up with a compiler.

A. Compilers and Interpreters

Unfortunately, there is no commonly accepted definition of the terms “compiler” and “interpreter”. The problem is the smooth transition between compilation and interpretation techniques, which blur a clear distinction. On the one hand many interpreters have integrated just-in-time compilers, on the other hand, some compilers rely on an interpreter integrated into each compiled unit. In combination with virtual machines [19], which have functionality not provided by any real machine, and CPUs, which can execute source code directly [20], it is even more complicated to find a clear distinction.

In our context, we differentiate between compiler and interpreter by the number of times *our* `ASTFactory` is called per JS-application execution. If the factory is called every time, the system is called interpreter. Otherwise, it’s a compiler.

B. Interpreter Design

In order to design, realize, and implement an interpreter based on an abstract syntax tree [21], current software engineering approaches recommend one of two main designs: the interpreter pattern and the visitor pattern [22].

According to the interpreter pattern, each node of the AST should have a specialized version of an evaluation, respectively, interpretation method; e.g., `eval(...)`. The visitor pattern in contrast only needs some callback functionality. In this way it can separate algorithms and actions from the data structure it operates on. As the visitor pattern (in combination with an iterator pattern for tree traversal) is already used by the Euclides compiler backends, it is also used by the interpreter.

The main idea of the interpreter implementation is based on a property found in many scripting languages. In contrast to, for example, Java, in which each statement is enclosed (at minimum) by a class definition, enclosed by a file definition, the scripting language JavaScript does not have this “overhead”. As a consequence, the root node of the AST is simply a list of statements: statementA, statementB, statementC and for each statement, the list of previous statements has to be a valid program. This linguistic property allows to compile each top-level JS statement as a unit of its own – a dynamic library. While this is not sensible for regular compilations, it offers the possibility to compile instructions statement by statement. Finally, if each unit is executed directly after being compiled, the resulting backend is an interpreter. Even more, additionally included callback routines can be used for debugging purposes [23].

C. Implementation Details

Following the observation that even a single statement can be regarded as a unit of its own, the original JavaScript compiler is extended to reflect this property. Statements in the AST are no longer stored in a one-dimensional array, but a two-dimensional array is used instead. This way it is possible to group statements, i.e., all statements passed to the interpreter in a single evaluation call form one group and are stored in a one-dimensional array. All groups are stored in an array as well, thus as a consequence, the statements are stored in a two-dimensional array. These groups can be accessed by a new set of access functions while at the

 ISBN: 978-1-61208-222-6

same time retain compatibility to the compiler, e.g., the command `getAllStatements()` now simply copies the two-dimensional structure in a one-dimensional one.

In addition to the changes in the AST, the namespace is also using a two-dimensional array for storing all symbols the same way the AST does. It uses the same mechanism to create units of symbols while being compatible to the old compiler version. These changes are necessary to allow tracking of interpretation history as well as to speed up all operations relying on the AST such as validation and code generation.

A small change in the runtime, not related to the interpreter redesign, was carried out in the process of implementing the changes for AST and namespace. Function pointers are now being ommited in the favor of using anonymous inner classes.

V. Conclusion

The simplicity of scripting languages reduces a user's inhibition threshold to start programming. Our generative modeling framework *Euclides* for non-expert users is based on a JavaScript dialect. It consists of a JavaScript compiler including a front-end (lexer, parser, etc.) and back-ends for several platforms.

The main contribution is an interactive interpreter based on the already existing compiler. Instead of creating large proportions of new code, whose behavior has to be consistent with the already existing compiler, we envisaged a minimally invasive solution. It allows us to reuse most parts of the compiler's front- and back-end.

Acknowledgment

We would like to thank Richard Bubel for his valuable support on ANTLR and the JS grammar. In addition, the authors gratefully acknowledge the generous support from the European Commission for the integrated project 3D-COFORM [24] under grant number FP7 ICT 231809 and from the Austrian Research Promotion Agency (FFG) for the research project METADESIGNER, grant number 820925/18236.

References

[1] J. K. Ousterhout, "Scripting: Higher Level Programming for the 21st Century," *IEEE Computer Magazine*, vol. 31, no. 3, pp. 23–30, 1998.

[2] M. Strobl, C. Schinko, T. Ullrich, and D. W. Fellner, "Euclides – A JavaScript to PostScript Translator," *Proccedings of the International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking (Computation Tools)*, vol. 1, pp. 14–21, 2010.

[3] C. Schinko, M. Strobl, T. Ullrich, and D. W. Fellner, "Modeling Procedural Knowledge – a generative modeler for cultural heritage," *Selected Readings in Computer Graphics 2010*, vol. 21, pp. 107–115, 2011.

[4] D. Brutzman, "The virtual reality modeling language and Java," *Communications of the ACM*, vol. 41, no. 6, pp. 57 – 64, 1998.

[5] J. Behr, P. Dähne, Y. Jung, and S. Webel, "Beyond the Web Browser – X3D and Immersive VR," *IEEE Virtual Reality Tutorial and Workshop Proceedings*, vol. 28, pp. 5–9, 2007.

[6] F. Breuel, R. Bernd, T. Ullrich, E. Eggeling, and D. W. Fellner, "Mate in 3D – Publishing Interactive Content in PDF3D," *Publishing in the Networked World: Transforming the Nature of Communication, Proceedings of the International Conference on Electronic Publishing*, vol. 15, pp. 110–119, 2011.

[7] M. Di Benedetto, F. Ponchio, F. Ganovelli, and R. Scopigno, "SpiderGL: a JavaScript 3D graphics library for next-generation WWW," *Proceedings of the 15th International Conference on Web 3D Technology*, vol. 15, pp. 165–174, 2010.

[8] T. Ullrich, C. Schinko, and D. W. Fellner, "Procedural Modeling in Theory and Practice," *Poster Proceedings of the 18th WSCG International Conference on Computer Graphics, Visualization and Computer Vision*, vol. 18, pp. 5–8, 2010.

[9] P. Prusinkiewicz and A. Lindenmayer, *The Algorithmic Beauty of Plants*, P. Prusinkiewicz and A. Lindenmayer, Eds. Springer-Verlag, 1990.

[10] Y. Parish and P. Mueller, "Procedural Modeling of Cities," *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, vol. 28, pp. 301–308, 2001.

[11] S. Havemann, "Generative Mesh Modeling," *PhD-Thesis, Technische Universität Braunschweig, Germany*, vol. 1, pp. 1–303, 2005.

[12] C. Reas, B. Fry, and J. Maeda, *Processing: A Programming Handbook for Visual Designers and Artists*, C. Reas, B. Fry, and J. Maeda, Eds. The MIT Press, 2007.

[13] C. Schinko, M. Strobl, T. Ullrich, and D. W. Fellner, "Modeling Procedural Knowledge – a generative modeler for cultural heritage," *Proceedings of EUROMED 2010 - Lecture Notes on Computer Science*, vol. 6436, pp. 153–165, 2010.

[14] ——, "Scripting Technology for Generative Modeling," *International Journal On Advances in Software*, vol. 4, pp. 308–326, 2011.

[15] T. Parr, *The Definite ANTLR Reference – Building Domain-Specific Languages*, T. Parr, Ed. The Pragmatic Bookshelf, Raleigh, 2007.

[16] E. Freeman, E. Freeman, B. Bates, and K. Sierra, *Head First Design Patterns*, E. Freeman, E. Freeman, B. Bates, and K. Sierra, Eds. O'Reilly Media, Inc., 2004.

[17] T. Ullrich and D. W. Fellner, "Generative Object Definition and Semantic Recognition," *Proceedings of the Eurographics Workshop on 3D Object Retrieval*, vol. 4, pp. 1–8, 2011.

[18] A. M. Ertl and D. Gregg, "Retargeting JIT compilers by using C-compiler generated executable code," *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, vol. 13, pp. 41–50, 2004.

 ISBN: 978-1-61208-222-6

[19] T. Lindholm and F. Yellin, *The Java(TM) Virtual Machine Specification*, T. Lindholm and F. Yellin, Eds. Prentice Hall, 1999.

[20] T. R. Bashkow, A. Sasson, and A. Kronfeld, "System Design of a FORTRAN Machine," *IEEE Transactions on Electronic Computers*, vol. 16, pp. 485–499, 1967.

[21] T. Parr, *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*, T. Parr, Ed. Pragmatic Bookshelf, 2010.

[22] M. Hills, P. Klint, T. van der Strom, and J. Vinju, "A Case of Visitor versus Interpreter Pattern," *Proceedings of the International Conference on Objects, Models, Components and Patterns (TOOLS'11)*, vol. 49, pp. 1–16, 2011.

[23] J. Vraný and A. Bergel, "The Debuggable Interpreter Design Pattern," *Proceedings of the International Conference on Software and Data Technologies*, vol. 2, pp. 22–29, 2007.

[24] D. Arnold, "3D-COFORM: Tools and Expertise for 3D Collection Formation," *Proceedings of Electronic Information, the Visual Arts and Beyond*, vol. 21, pp. 94 – 99, 2009.

 ISBN: 978-1-61208-222-6