Scripting Technology for Generative Modeling

Christoph Schinko[¶], Martin Strobl[¶], Torsten Ullrich[§], and Dieter W. Fellner[¶‡]

[¶] *Institut für ComputerGraphik & WissensVisualisierung, Technische Universität Graz, Austria*
c.schinko@cgv.tugraz.at, m.strobl@cgv.tugraz.at
[§] *Fraunhofer Austria Research GmbH, Graz, Austria*
torsten.ullrich@fraunhofer.at
[‡] *GRIS, TU Darmstadt & Fraunhofer IGD, Darmstadt, Germany*
d.fellner@igd.fraunhofer.de

***Abstract*—In the context of computer graphics, a generative model is the description of a three-dimensional shape: Each class of objects is represented by one algorithm M. Furthermore, each described object is a set of high-level parameters x, which reproduces the object, if an interpreter evaluates $M(x)$. This procedural knowledge differs from other kinds of knowledge, such as declarative knowledge, in a significant way. Generative models are designed by programming. In order to make generative modeling accessible to non-computer scientists, we created a generative modeling framework based on the easy-to-use scripting language *JavaScript* (JS). Furthermore, we did not implement yet another interpreter, but a JS-translator and compiler. As a consequence, our framework can translate generative models from *JavaScript* to various platforms. In this paper we present an overview of *Euclides* and quintessential examples of supported platforms: Java, Differential Java, and GML. Java is a target language, because all frontend and framework components are written in Java making it easier to be embedded in an integrated development environment. The Differential Java backend can compute derivatives of functions, which is a necessary task in many applications of scientific computing, e.g., validating reconstruction and fitting results of laser scanned surfaces. The postfix notation of GML is very similar to that of Adobes Postscript. It allows the creation of high-level shape operators from low-level shape operators. The GML serves as a platform for a number of applications because it is extensible and comes with an integrated visualization engine. This innovative meta-modeler concept allows a user to export generative models to other platforms without losing its main feature – the procedural paradigm. In contrast to other modelers, the source code does not need to be interpreted or unfolded, it is translated. Therefore, it can still be a very compact representation of a complex model.**

***Keywords*-Generative modeling, procedural modeling, computer graphics, JavaScript, compiler**

I. Introduction

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 have gained a lot of attention in the past few years.

In the context of Cultural Heritage, the Generative-Modeling-Language (GML) is an established procedural modeling environment designed for expert users. The aim of the *Euclides* modeling framework 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 [1].

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 [2] and X3D [3] files. It checks user forms in PDF files [4], controls game engines [5], configures applications, and performs many more tasks. According to J. K. Ousterhout scripting languages use a higher level of abstraction compared to system programming languages as they are often typeless and interpreted to emphasize the rapid application development purpose [6]. Whereas system programming languages are designed for creating algorithms and data structures based on low-level data types and memory operations. As a consequence, graphics libraries [7], graphics shaders [8] and scene graph systems [9], [10] are usually written in C/C++ dialects [11], and procedural modeling frameworks use scripting languages such as Lua, JavaScript, etc.

A. Geometric Modeling

When describing the shape of three-dimensional objects, two different approaches are established:

- composing an object of basic primitives (points, triangles, quads, etc.),
- creating a procedural description [12].

A composition of primitives can be achieved by conventional geometric modeling or by using 3D acquisition devices, which are always more or less noisy. Whereas, a procedural description is based on an ideal object rather than a real one and is often used to describe an object's inherent properties. Its strength lies in a very compact description, which, compared to conventional geometric descriptions, is not dependent on the number of primitives but on the

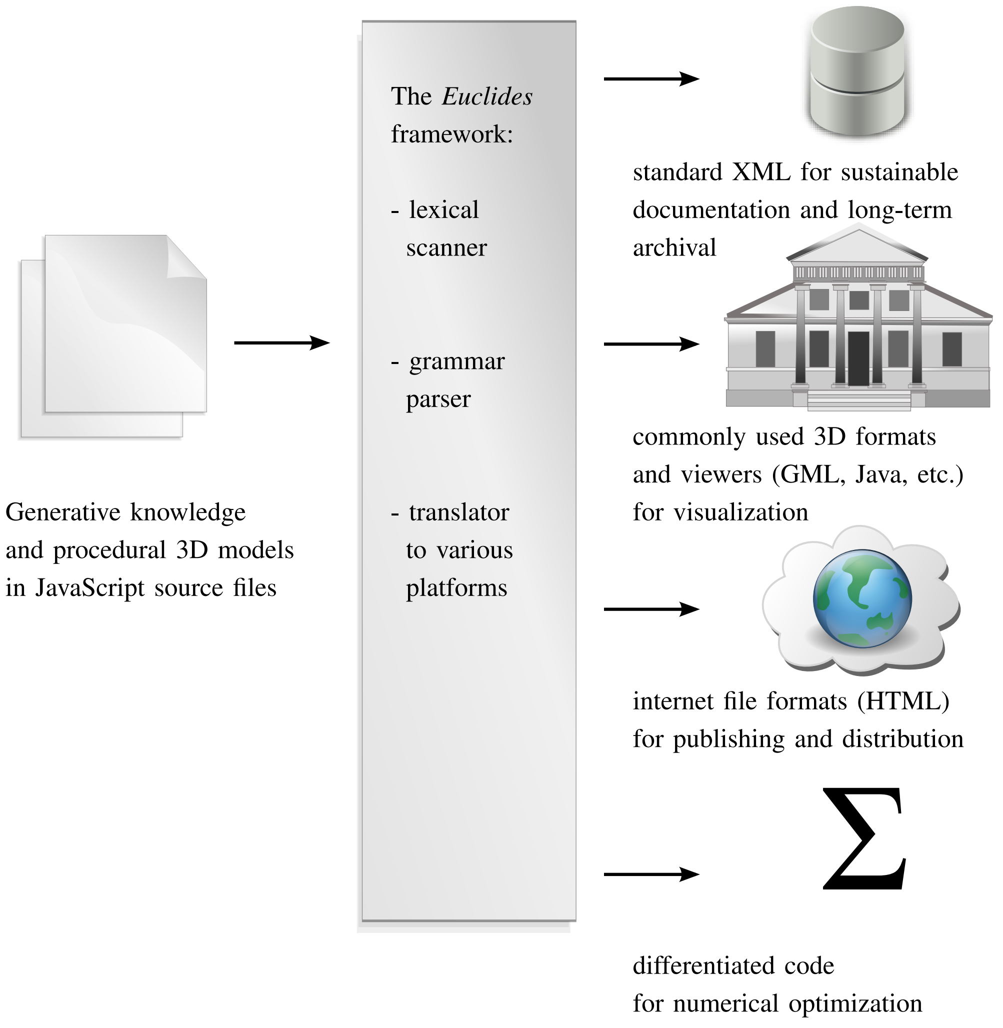

The *Euclides* framework:
- lexical scanner
- grammar parser
- translator to various platforms
Generative knowledge and procedural 3D models in JavaScript source files
standard XML for sustainable documentation and long-term archival
commonly used 3D formats and viewers (GML, Java, etc.) for visualization
internet file formats (HTML) for publishing and distribution
Σ
differentiated code for numerical optimization

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*.

model's complexity itself. However, generative models must not be seen as a replacement for established geometric descriptions, but as a semantic enrichment. Another advantage of procedural 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, thus making the object easily identifiable by digital library services (indexing, markup and retrieval). For a specific object only its type and its instantiation parameters have to be identified. In combination, these two methods can be used to perform detailed mesh comparisons, which can reveal the smallest changes and damage of digitized artifacts. Such analysis and documentation tasks are valuable in the context of cultural heritage [13], [14].

B. Cultural Heritage

Procedural modeling techniques are well studied in the fields of computer-aided design and engineering. Unfortunately, in the context of cultural heritage, model complexity, model size, and imperfection have dimensions several orders of magnitude higher than many other fields of applications [15]. Cultural heritage artifacts often have a high inherent complexity, since they represent masterpieces of the human creative genius. As such artifacts are seldom single findings, they are therefore often embedded in larger excavation sites. An archaeological excavation may have an extent on the scale of kilometers with a high richness of detail on the scale of millimeters. Domain knowledge by cultural heritage experts and procedural modeling techniques are keys to cope with this complexity and size [16].

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. This is why we use JavaScript – a beginner friendly, structured language.

C. JavaScript

JavaScript features a rather intuitive syntax, which is easy to read and to understand. A comprehensible, well-arranged syntax is useful, since source code is more often read than written. JavaScript supports features like dynamic typing and first-class functions. The most important feature is, that it is wide-spread amongst non-computer scientists – namely designers and creative coders. This is the reason why there are numerous tutorials available on the internet, resulting in an easy access to the language. JavaScript is used in many different environments and has evolved from being used only in a web-browser to a flexible multi-purpose scripting language. Our integrated scripting solution adds another chapter to the history of JavaScript usage.

Our meta-modeler approach *Euclides* is based on JavaScript and differs from other modeling environments in a very important aspect: target independence. Usually, a generative modeling environment consists of a script interpreter and a 3D rendering engine. A generative model (3D data structures with functionality) is interpreted directly to generate geometry, which is afterwards visualized by an integrated rendering engine.

In our system a model's source code is not interpreted but parsed into an intermediate representation, an abstract syntax tree (AST). After a validation process it is translated into a target language. The process of

$$\textbf{parsing} \rightarrow \textbf{validating} \rightarrow \textbf{translating}$$

offers many advantages as illustrated in Figure 1. The validation step involves syntax and consistency checks. These checks are performed to ensure the generation of a correct intermediate representation and to provide meaningful error messages as early as possible within the processing pipeline. The translation step, like every compilation/translation (see Section II, Related Work), consists of a parser frontend (see Section III, JavaScript Frontend), middleware, and backend (see Section IV, Target Backends), and offers platform independence (see Section V, Conclusion and Future Work). The same code basis can be translated into different languages for various purposes.

II. Related Work

A. Generative Modeling

Procedural modeling systems often rely on grammars to describe the rules behind generative components. Early systems based on grammars were Lindenmayer systems [17], or L-systems for short. These systems provide the means for modeling plants, where they were successfully applied. Starting with simple strings, complex strings are created by using a set of string rewriting rules. A predefined set of rules is applied to an initial string forming a new, possibly larger string. The L-systems approach reflects a biological motivation. In order to use L-systems to model geometry an interpretation of the generated strings is necessary. The modeling power of these early geometric interpretations of L-systems was limited to creating fractals and plant-like branching structures. This leads to the introduction of parametric L-systems. The idea is to associate numerical parameters with L-system symbols to address continuous phenomena, which were not covered satisfactorily by L-systems alone.

CGA Shape, CityEngine: L-systems in combination with shape grammars are successfully used in procedural modeling of cities [18]. Parish and Müller presented a system that generates a street map including geometry for buildings given a number of image maps as input. For that purpose L-systems have been extended to allow the definition of global objectives as well as local constraints. However, the use of procedurally generated textures to represent facades of buildings often results in a limited level of detail. In later work, Müller et al. describe a system [19] to create more detailed facades based on a split grammar called *CGA shape*. A framework called the *CityEngine* provides a modeling environment for *CGA shape*. It relies on different views to guide an iterative modeling process.

Lipp et al. presented another modeling approach [20] following the notation of Müller [21] that deals with the aspects of more direct local control of the underlying grammar by introducing visual editing. The idea is to modify elements selected directly in a 3D-view, rather than editing rules in a text based environment. Principles of semantic and geometric selection are therefore combined as well as functionality to store local changes persistently over global modifications.

Model Graphs: A modeling method as well as a graphical user interface for the creation of natural branching structures was proposed by Lintermann et al. [22]. The idea is to represents the modeling process with a structure tree, which can be altered using specialized components describing geometry as well as structure. Another set of components can be used for defining global and partial constraints. These components are described procedurally using creation rules, which include recursion. Geometric data is generated according to the structure tree via a tree traversal, where the components generate their geometrical output themselves.

Ganster et al. propose a procedural modeling approach [23] based on structure trees as well. They describe an integrated framework relying on a visual language. The infix notation of the language requires the use of variables, which are stored on a heap. A graph structure represents the rules used to create an object. Special nodes allow the creation of geometry, the application of operators as well as the usage of control structures. Various attributes can be set for nodes used in a graph. Directed edges between nodes define the order of execution, in contrast to a visual data flow pipeline where data is transported between the different stages.

Hierarchical Description: Finkenzeller presented another approach for detailed building facades [24] called *ProcMod*. It features a hierarchical description for an entire building. In order to create a building, the user provides a coarse outline and a basic style of the building including distinguished parts. The system then generates a graph representing the building. In the next step, the graph is traversed and geometry for every element of the graph is generated. This results in a detailed scene graph, in which each element can be modified afterwards. The version described has some limitations: for example, organic structures and inclined walls cannot be modeled.

Postfix Expressions: Havemann proposes a stack based language called *Generative Modeling Language* (GML), which allows, but is not limited to, creating polygonal meshes [25]. The postfix notation of the language is very similar to that of *Adobe Postscript*. High-level shape operators are created from low-level shape functionality. The GML serves as a platform for a number of applications, because it is extensible and comes with an integrated visualization engine.

An extended system presented by Mendez et al. combines semantic scene-graph markups with generative modeling [26]. The purpose of the system is the generation of semantic three dimensional models of underground infrastructure. A geospatial database and a rendering engine are combined in order to create an interactive application. The GML is used for on-the-fly generation of procedural models in combination with a conventional scene graph system with semantic markup.

Scripted Modelers: In contrast to specialized generative modelers, there are a number of 3D modeling software packages available like Autodesk Maya^{TM} or 3ds Max^{TM}. They provide a variety of tools for modeling with polygons, non-uniform rational B-splines (NURBS) and predefined primitives. In addition to a graphical user interface (GUI), a scripting language is supplied to extend the functionality. It enables tasks that cannot be achieved easily using the GUI and speeds up complicated or repetitive tasks.

Processing and Grasshopper: Processing stands for a programming language and a development environment. It was initially created to serve as a software sketchbook and to teach students fundamentals of computer programming [27]. It quickly developed into a tool that is used for creating visual arts. Processing is basically a Java-like interpreter offering a new graphics and utility API together with some usability simplifications. A large community behind the tool produced over seventy libraries to facilitate computer vision, data visualization, music, networking, and electronics.

Another tool with a creative background is Grasshopper [28]. Its main purpose is the creation of graphical algorithms. It is a graphical editor for Rhino's 3D modeling tools designed to be used without programming skills - unlike RhinoScript. In Grasshopper programs are created by dragging components onto a canvas and interconnecting these components. Many components create, but are not limited to, 3D geometry. Similar to Processing, there is a large community behind the tool.

B. *JavaScript*

JavaScript started as a simple client-side scripting language. Nowadays, there are a number of projects that use JavaScript in innovative ways. EMScripten (https://github.com/kripken/emscripten) is a LLVM to JavaScript compiler. LLVM stands for low level virtual machine, and is an intermediary representation for code compiled from languages such as C, C++ or Objective-C. LLVM output used by EMScripten is similar to assembly language.

The difficulty in translating an LLVM AST to JavaScript is that the high level representation of the source languages is lost. A translator hence needs to compact ambiguous assembly statements into general, high level language code. The EMScripten algorithm, called relooper, produces output that models a virtual machine. The heap of the virtual machine is a huge array. A growing machine's stack is modeled as a variable that serves as an index to the heap. Necessary control flow such as `goto` and arbitrary labels are modeled as looping switch-statements. EMScripten author Zakai is demonstrating the versatility of the transpiler by porting the computer game DOOM to JavaScript. The JavaScript output of compiled DOOM is accelerated by the optimizing compiler *GClosure* by *Google*.

Similar to a virtual machine, Bellard has written a PC emulator that runs in JavaScript. It uses JavaScript's typed arrays to emulate a feature-stripped 486 CPU. Typed arrays allow to apply views on an existing array, by which a programmer can access the array contents as differently sized chunks. This way, an array of 32 bit sized numbers might be accessed as 16 bit array. This way of addressing allows extracting single bytes from an array of integers. Obviously, bit-level handling is possible and facilitated, if one uses a byte to represent a machine-bit, scaling available memory by a factor of 8. To demonstrate the power of the emulator, Bellard shows a version of a small GNU/Linux distribution that is properly executed by his JavaScript-based PC emulator.

III. JavaScript Frontend

A. Lexer and Parser

Despite the name, JavaScript is unrelated to the programming language Java, even if it copies many names and naming conventions. It is a functional programming language with support of structured programming constructs in C-style (e.g., if-statements, for-loops, switch-statements). In analogy to C, JavaScript differentiates between expressions and statements. However, there are a few important aspects that need to be mentioned:

- In contrast to C-style block-level scoping, JavaScript supports function-level scoping.
- Types are dynamic and they are associated with values; i.e., a variable's value defines its type.
- Functions are objects themselves and therefore can be assigned to variables, returned by functions, passed as arguments, and manipulated like any other object [29].

As JavaScript is typically interpreted, its design reflects interactivity. It relies on a run-time environment, i.e., an object model provides the functionality to communicate with the host environment. Furthermore, the default entry point for a parser is a *statement* rule as visualized in Figure 2. It does not have a mandatory, enclosing class structure or a main function as entry point. Additionally, the interactivity is reflected in required forward declarations, which can be created via forward references, as functions are first-class citizens.

The *statement* rule is split up in several sub-rules like *statementIf*, *statementDoWhile*, *statementExpression*. While these rules are rather straightforward, the *statementNativeCode* rule is a special feature of our grammar. It allows to embed native code into JavaScript. When JavaScript code gets translated, it is sometimes necessary to embed code written in the target language – so called native code. As a consequence, each platform-dependent library is available in all target versions. An illustrating example is taken from the Java version of the *Euclides*' *IO* library. It shows how writing output to the text console is handled.

The function `io_stdout_write` writes a text message to Standard Out. If necessary the parameter `msg` will be converted to a string. In order to be JavaScript compliant, native code is embedded in comments using the special character sequence `/*%    */` to denote the beginning of a native code section. A similar mechanism is used to allow parsing annotations, where the special character sequence `/*@    */` is used to denote an annotations section inside a comment. Annotations are used similar to preprocessor directives in C. They are evaluated by the parser and additionally embedded in the target source code.

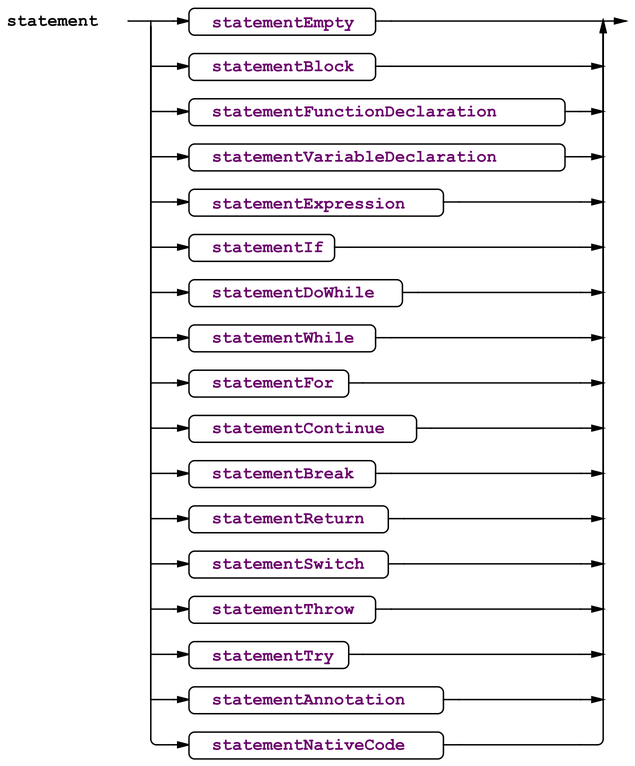


Figure 2. The entry point for a JavaScript parser is a *statement* rule, because JavaScript is typically interpreted in a run-time environment – statement by statement. As a consequence, the JavaScript grammar does not contain any enclosing class structures.

```
/*@
  euclides.suppress_warning_unreferenced_o ↩
bject.io_stdout_write;
*/

/**
  This function writes a text message to
  standard out. The parameter 'msg' will be
  converted to a 'string', if necessary.
*/
function io_stdout_write(msg)
{
  /*%
  System.out.print(usr_msg.toString());
  */
}
```

In the example above, the annotations technique is used to suppress a warning of an unreferenced object, which might occur, if the function is not used.

B. Abstract Syntax Tree

Our parser for JavaScript is written using ANother Tool for Language Recognition (ANTLR). ANTLR provides a framework for constructing recognizers, interpreters, compilers, and translators from grammatical descriptions [30]. It introduces a strategy called *LL(*)* parsing, which extends the *LL(k)* parsing strategy with lookahead of arbitrary length without explicitly specifying it. The purpose of using this framework is to syntactically and semantically check the provided input for JavaScript compliance and at the same time to generate an intermediate representation: an abstract syntax tree.

The AST offers three entry points into a script. The first entry point is the "obvious" representation: a sequence of statements. Each statement contains all included substatements and expressions as well as associated comments. The leaves of the AST store tokens together with formatting information (line and position). This tree structure is extended by reference and occurrence links; e.g., each method call references the method definition and each variable definition links to all its occurrences.

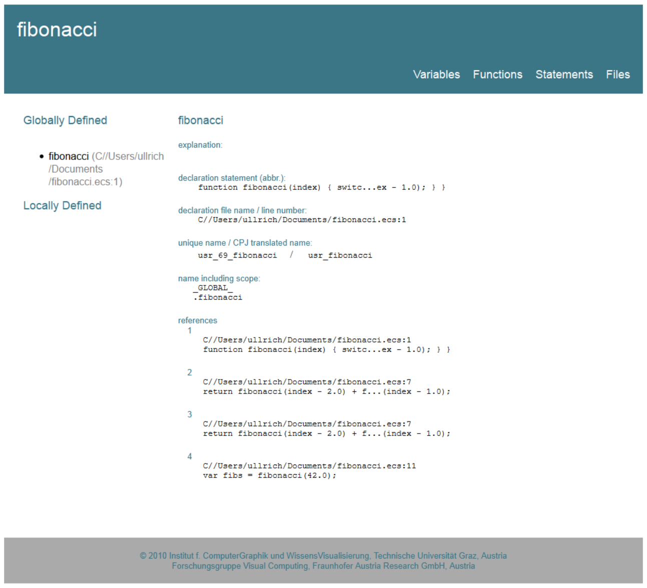
The list of all variables represents a second entry point to explore the AST. Each entry consists of the variable name, the statement in which it is defined, the scope of the variable as well as all occurrences in the source code.

Last but not least, the third entry point lists all function implementations including anonymous functions with complete function body. Similar to the contents of the variable list, each entry offers the function name, its defining statement, the scope of the function, the number of parameters this function takes, as well as all occurrences in the source code.

These structures are not only needed during the translation process, but they are valueable inspection tools. *Euclides*' automatic documentation system exports these views and data structures in a collection of XHTML files: Using markup techniques directly jumping between occurrences and definitions of variables, function, etc. is possible; e.g., a screenshot of the automatic documentation created for the fibonacci example

```
function fibonacci(index) {
  switch (index) {
    case 0:
    case 1:
      return 1;
    default:
      return fibonacci(index-2)
           + fibonacci(index-1);
  }
}

var fibs = fibonacci(42);
```



```
fibonacci

explanation:

declaration statement (abbr.):
    function fibonacci(index) { switc...ex - 1.0); } }

declaration file name / line number:
    C//Users/ullrich/Documents/fibonacci.ecs:1

unique name / CPJ translated name:
    usr_69_fibonacci  /  usr_fibonacci

name including scope:
  _GLOBAL_
  .fibonacci

references
  1
    C//Users/ullrich/Documents/fibonacci.ecs:1
    function fibonacci(index) { switc...ex - 1.0); } }

  2
    C//Users/ullrich/Documents/fibonacci.ecs:7
    return fibonacci(index - 2.0) + f...(index - 1.0);

  3
    C//Users/ullrich/Documents/fibonacci.ecs:7
    return fibonacci(index - 2.0) + f...(index - 1.0);

  4
    C//Users/ullrich/Documents/fibonacci.ecs:11
    var fibs = fibonacci(42.0);
```

• fibonacci (C//Users/ullrich/Documents/fibonacci.ecs:1)

© 2010 Institut f. ComputerGraphik und WissensVisualisierung, Technische Universität Graz, Austria
Forschungsgruppe Visual Computing, Fraunhofer Austria Research GmbH, Austria

Figure 3. The *Euclides* documentation target represents JavaScript as a sustainable, standard-conform XML document can be displayed in an arbitrary web browser.

is shown in Figure 3. All globally and locally defined variables are listed in the Variables view. Several properties are available for each variable:

- Comments. Any comments associated with a variable are preserved and included.
- Location. The line of code (source, its line number and file name) where the variable is declared.
- Visibility. The name together with the scope, in which the variable is available.
- References. All references and uses of the variable in the source code including file name, line number and declaration statement.

Similarly to the Variables view, the Functions view is a collection of all functions defined in the source code and consists of the same four properties mentioned above. The Statements view is a collection of all statements of the source code together with filename and line number, which allows, for example, identifying duplicate code snippets. Also it gives a nice overview of the complexity of the source code. In the files view, the source code is available as XML document.

C. Libraries

A collection of libraries for geometry (*GEO*), graphical user interfaces (*GUI*), input/output (*IO*), mathematics (*MATH*) and some utilities (*UTL*) are available for the *Euclides* framework. They offer functionality to conveniently create generative models together with basic user interface elements. A simple example visualization of a Sierpinski tetrahedron with GUI components to control the subdivision depth is shown in Figure 4.

IV. Target Backends

When translating source code into target language code [31], the need to establish a proper naming standard quickly arises. A runtime environment is implemented using symbols and constructs available in the target language. Naming these constructs may interfere with the naming of code to be translated. For example, there may very likely be a function called `main` in the runtime environment rendering this name to be reserved. In order to overcome these limitations, all names in the translated code are extended with the prefix `usr_`, if it corresponds one-to-one to a name in JavaScript. Otherwise it is prefixed with `sys_`.

Additionally, dealing with different target languages not only means dealing with naming issues, but also dealing with character encoding. A character may be allowed to be used in JavaScript, but forbidden in a target language. As a consequence, we introduced lists of allowed characters and rules for character replacements to handle all naming issues.

These two mechanisms ensure the validity and consistency concerning naming and encoding issues of the generated code. The result of a name translation is always a name, which is

1) valid in the target language and
2) does not collide with any predefined names, name spaces, or keywords.

A. Java

Although Java and JavaScript have some similarities, the concepts of both languages show major differences. Java is a statically typed, class-based, general-purpose programming

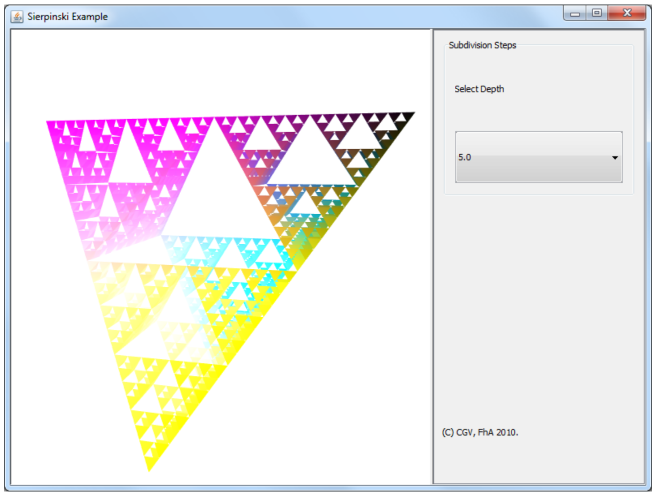


Figure 4. This figure shows how a Sierpinski tetrahedron example translated to the Java target looks like. A small GUI with a drop-down list to select the subdivision depth together with a 3D view of the Sierpinski tetrahedron at subdivision depth five is scripted using *Euclides*.

language designed to have a minimum of implementation dependencies to be able to follow the credo: “write once, run anywhere”.

It is chosen as a target language, because all frontend and framework components are written in Java making it easier to be embedded in an integrated development environment.

Data Types: Because of conceptual differences in the typing system, it is unpractical to project JavaScript data types onto built-in Java data types. For example, JavaScript makes no difference between integer numbers or floating-point numbers. There is just one data type called `number` that may hold any type of number. Similar difference can be found when comparing the remaining data types.

Dynamic typing is another big difference between the two languages. As a consequence, each JavaScript data type is re-built in Java to match its functionality making a total of seven data types. These data types are wrapped in a class called `Var`, which provides the properties,

- `getType()`
- `length(int ii)`

access functions

- `accessArray(int ii, Var index)`
- `accessObject(int ii, String attribute)`
- `assign(int ii, Var variable)`
- `delete(int ii, Var variable)`
- `executeDirect(int ii, Var THIS, Var[] parameters)`
- `executeIndirect(int ii, String attribute, Var[] parameters)`

and conversion methods applicable to all JavaScript variables:

- `toArray()`
- `toBoolean()`
- `toFunction()`
- `toNumber()`
- `toObject()`
- `toString()`
- `toUndefined()`

In these methods the parameter `ii` always refers to a table entry, which references the corresponding line of JavaScript source code; e.g. each data type can be accessed like an array. In case of an array, the access is “as supposed”, in case of a String it is character-wise, in all other cases an implicit conversion creates a new, empty array. As our runtime environment produces warnings, if implicit conversion take place, the implementation of an array access includes the statement `Log.variableTypeChangeImplicit(ii);`. In the messages table (generated by the compiler) entry #`ii` references information needed for a reasonable warning; e.g. during the execution of

```
var number = 42;
number = "Hello World";
```

the runtime environment produces the warning

```
assignment provoked a warning.
  type    : variable type change by assignment
  file    : C//Users/ullrich/warning.ecs
  line    : 2
  details : number = "Hello World";
```

The access functions reveal the implementation details and the mappings of the Java types.

Boolean: The corresponding Java 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 `ArrayList<Var>`.

Object: And an `object` in JavaScript is mapped to `HashMap<String,Var>`.

Function: The corresponding object to a JavaScript functor is a function pointer implementation in Java via abstract objects.

With these data types comes the necessity to use a runtime environment in the translated Java code. Whenever a variable is created or a value is assigned, a method-call is performed – thus significantly increasing the execution time of the code. However, for creating variables, a factory pattern is applied with the inherent advantage of exchangeability. This design pattern is extensively used by the "Differential Java" backend, which is described in the next section.

Concerning language constructs a wide range can be translated easily, since they have the same semantic meaning in both languages. Sometimes, there is the need to utilize temporary variables, which implicate a possible naming conflict with variable names used in the original JavaScript source code. This problem is tackled by prefixing all original JavaScript names and additionally creating unique names for temporary variables as mentioned before.

Functions: In Java, invokable routines are called methods and they are similar to, but not quite like functions in JavaScript. The runtime environment provides a class for JavaScript functions to mimic their behavior. An important property of functions in JavaScript is that they can be `undefined`. Therefore, when instantiating an empty function in Java, a *dummy* with the correct behavior is returned. Executing a function in the Java runtime environment is done by calling the `execute` method in the function class. In addition to function parameters, an environment reference is passed to the function in order to enable correct interaction with the immediate environment. Functions extend an abstract class called `Fct` defining all necessary methods:

- `getID()`
- `getName()`
- `getTranslatedName()`
- `getAnnotations()`
- `getParam()`
- `getParams()`
- `execute(int ii, Var THIS, Var[] parameters)`
- `execute(int ii, Var THIS, Var usr_vecArray)`

They reside in a *public*, *final* class called `Function`. Consequently, the function

```
function add(a, b) {
  return a + b;
}
```

gets translated to

```
@Override
public Var execute(int ii, Var THIS,
                   Var usr_a, Var usr_b) {
  try {
    {
      if (Main.AVOID_UNREACHABLE_CODE_ERROR)
        return Op.ADD(0, usr_a, usr_b);
    }
  } catch (EuclidesRuntimeException exp) {
    throw exp;
  } catch (RuntimeException exp) {
    Log.uncaughtException(ii);
    System.err.println(exp);
    System.exit(0);
  }
  return Factory.initUndefined();
}
```

The body of the function is embedded in a try-catch block in order to throw runtime exceptions or halt execution in case of an unhandled exception. The value `undefined` is returned in case a runtime exception occurs. Please note, the static constant `Main.AVOID_UNREACHABLE_CODE_ERROR` is always true and only needed to avoid – as it says – "unreachable code errors" thrown by Java compilers, for example, if a return-statement is followed by further statements.

Translated functions and parameters are named just like their JavaScript-counterparts (except for the `usr_` prefix).

Operators: Since JavaScript data types are not mapped to native Java data types, all operators need to be recreated in the Java runtime environment as well. A total of 35 operators grouped in unary, binary and tertiary operators are available. Since each operator is applied via a method call, they can be easily exchanged. Operators are collected as methods in a *public*, *final* class called `Op`. As an example, the following operation

```
var c = 19.0 + 23.0;
```

results in

```
Variable.usr_c.assign(1, Op.ADD(0,
Factory.initNumber(19.0),
Factory.initNumber(23.0)));
```

The result of the call to `Op.ADD` with the two numbers as parameters is stored in a new variable, which is returned and then used as a parameter for the assignment operation.

Control Flow: Control flow statements are widely identical in both languages. One of the differences, however, is the switch-statement. For a switch-statement in Java only primitive data types are allowed, whereas JavaScript allows all types to be used, attributable to dynamic typing. In order to obtain a correct translation, the switch-statement needs to be rewritten, which is done directly in the translated code. The first step is to analyze the statement from back to front comparing each case with the switching expression. Then the result is stored in a temporary variable and the switch-statement is rebuilt in reverse order using the temporary variable as switching expression. As a result

```
switch (favoritelanguage) {
  case "Java":
    io_stdout_write("Good choice!");
    break;
  case "C":
    io_stdout_write("Bad choice");
    break;
  default:
    io_stdout_write("I have no idea");
}
```

becomes

```
int sys_42 = 0;
if (Op.EQ(9, Variable.usr_favoritelanguage,
  Factory.initString("C")).toBoolean())
  sys_42 = 1;
if (Op.EQ(10, Variable.usr_favoritelanguage,
  Factory.initString("Java")).toBoolean())
  sys_42 = 2;
switch(sys_42) {
case 2:
  Function.usr_io_stdout_write.execute(11,
  THIS, Factory.initString("Good choice!"));
  if (Main.AVOID_UNREACHABLE_CODE_ERROR) break;
case 1:
  Function.usr_io_stdout_write.execute(12,
  THIS, Factory.initString("Bad choice"));
  if (Main.AVOID_UNREACHABLE_CODE_ERROR) break;
default:
  Function.usr_io_stdout_write.execute(13,
  THIS, Factory.initString("I have no idea"));
}
```

The corresponding translation in Java creates the temporary variable `sys_42` for comparisons and a switch-statement in reverse order to rebuild the behavior of the JavaScript counterpart.

Once all target files containing source code are generated, they are compiled using the Java compiler included in Java Platform, Standard Edition (Java SE). The resulting class files are automatically packed into a single JAR file for easy execution. As a last step, the JAR file is digitally signed to be ready-to-use for Java Web Start. The signature information becomes part of the embedded manifest file.

B. *Differential Java*

Besides the previously described Java target, *Euclides* offers a Differential Java backend. Computing derivatives of functions is a necessary task in many applications of scientific computing, e.g. validating reconstruction and fitting results of laser scanned surfaces [32], [33]: In combination with variance analysis techniques, generative descriptions can be used to validate reconstructions. Detailed mesh comparisons can reveal smallest changes and damages. These analysis and documentation tasks are needed not only in the context of cultural heritage but also in engineering and manufacturing. The *Euclides* framework is used to implement generative models, whose accuracy and systematics describe the semantic properties of an object; whereas the actual object is a real-world data set (laser scan or photogrammetric reconstruction) without any additional semantic information.

This analysis task needs derivatives of the distance-based objective function as well as the embedded procedural descriptions. According to Hammer et al. [34] there are three different methods to obtain values of derivatives:

- Numerical differentiation uses difference approximations to compute approximations of the derivative values.
- Symbolic differentiation computes explicit formulas for the derivative functions by applying differentiation rules.
- Automatic differentiation also uses the well-known differentiation rules, but it propagates numerical values for the derivatives.

Automatic differentiation combines the advantages of symbolic and numerical differentiation [35]. There are two important things to mention:

- Numbers instead of symbolic formulas must be handled.
- The computation of the derivative values is done automatically together with the computation of the function value.

Automatic differentiation evaluates functions specified by algorithms or formulas. All operations are performed according to the rules of a differentiation arithmetic given by “C++ for Verified Computing” [34]. First order differentiation

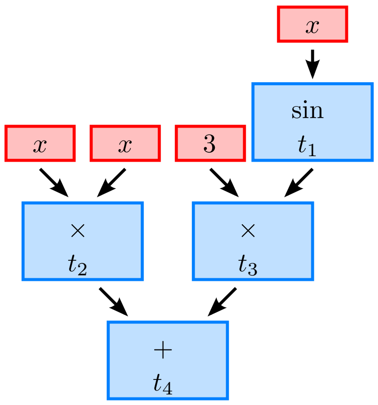


Figure 5. The evaluation of the term $x^2 + 3\sin x$ at $x_0 = 1.3$ using differentiation arithmetic does not only return its value but also its derivative value. The computational complexity of this differentiation arithmetic (forward method) is at most a small multiple of the cost of evaluating the term itself.

arithmetic is an arithmetic for ordered pairs in the one-dimensional case: the first component contains the value $u(x)$ of the function $u : \mathbb{R} \to \mathbb{R}$ at the point $x \in \mathbb{R}$. The second component contains the value of the derivative $u'(x)$. Familiar rules of calculus are used in the second component. The operations in these definitions are operations on real numbers.

An independent variable x and the arbitrary constant c correspond to the ordered pairs

$$(x, 1) \text{ and } (c, 0), \tag{1}$$

since $\frac{dx}{dx} = 1$, and $\frac{dc}{dx} = 0$. If the independent variable x of a formula for a function $f : \mathbb{R} \to \mathbb{R}$ is replaced by $X = (x, 1)$, and if all constants are replaced by their $(c, 0)$ representation, then the evaluation of f using the rules of differentiation arithmetic gives the ordered pair

$$\begin{aligned} f(X) &= f((x, 1)) && (2)\\ &= (f(x), f'(x)). && (3) \end{aligned}$$

For example, Figure 5 shows an AST whose evaluation at $x_0 = 1.3$ illustrates the calculation of its derivative values at intermediate subterms. For elementary functions

$$s : \mathbb{R} \to \mathbb{R} \tag{4}$$

the rules of differentiation arithmetic must be extended using the chain rule

$$\begin{aligned} s(U) &= s((u, u')) && (5)\\ &= (s(u), u' \cdot s'(u)). && (6) \end{aligned}$$

This way the sine function is defined by

$$\begin{aligned} \sin U &= \sin(u, u') && (7)\\ &= (\sin u, u' \cdot \cos u). && (8) \end{aligned}$$

The result of this structure and its corresponding operators is the algebra of dual numbers [36], which can be implemented in three ways:

Many programming languages offer an overloading mechanism that replaces each real number by a pair of real numbers including the differential. Each elementary operation on real numbers is overloaded, i.e., internally replaced by a new one, working on pairs of reals, that computes the value and its differential. In this way the original program is virtually unchanged.

Another approach uses source code transformation. This technique adds new variables, arrays, and data structures into the program that will hold the derivatives and the new instructions that compute them. This approach does not depend on language features such as operator overloading.

The third way to implement automatic differentiation does not modify a program or its source, but the platform (e.g. Java Virtual Machine, .Net Common Language Runtime, etc.) it runs on.

The Java differential target uses the third approach to automatically obtain derivatives. This is done by replacing variables and operators in the runtime environment, which is an easy task, since variables and operators are created using the factory pattern. The following listing shows the differences between standard and differential multiplication operator. As expected, the standard operator returns a variable initialized with the result of the multiplication operation.

```
/**
 * Binary operator multiply.
 *
 * @param ii Information index.
 * @param v1 The first operand.
 * @param v2 The second operand.
 * @return The result.
 */
public static Var MUL(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());
}
```

The differential operator calculates the derivatives of the operands and stores them in an array. Then a resulting array is constructed out of the calculated derivatives and returned as a variable.

```
/**
 * Binary operator multiply.
 *
 * @param ii Information index.
 * @param v1 The first operand.
 * @param v2 The second operand.
 * @return The result.
 */
public static Var MUL(int ii, Var v1, Var v2)
{
  if (!v1.getType().equals(Type.NUMBER)
   || !v2.getType().equals(Type.NUMBER))
    Log.deviantOperatorCallNoNumber(ii);

  double[] d1 = v1.toDifferential();
  double[] d2 = v2.toDifferential();
  double[] r = Factory.differential();
  r[0] = d1[0] * d2[0];
  for(int i=1; i<r.length; i++)
    r[i] = d1[i]*d2[0] + d1[0]*d2[i];

  return Factory.initNumber(r);
}
```

C. GML

The Generative-Modeling-Language (GML) is a procedural modeling environment predominantly used in the context of Cultural Heritage [37]. The corresponding translation mechanism 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 [1].

Data Types: In JavaScript each variable has a particular, dynamic type. It may be `undefined`, `boolean`, `number`, `string`, `array`, `object`, or `function`. GML also has a dynamical type system. Unfortunately, both type systems are incompatible to each other. Therefore, translating JavaScript data types to GML poses two particular problems: On the one hand, the dynamic types must be inferred at run time. On the other hand, GML's native data types lack distinct features needed by JavaScript. GML-Strings, for example, cannot be accessed character-wise. We solved these problems by implementing JavaScript-variables as dictionaries [25] in GML. Dictionaries are objects that map unique keys to values. These dictionaries hold needed metadata and type information as well as methods, which emulate JavaScript behavior. As we will show later, we will utilize GML's dictionaries for scoping as well.

The system translation library for GML, which every JavaScript-translated GML program defines prior to actual program code, contains the function `sys_init_data`, which defines an anonymous data value in the sense of JavaScript data.

```
/sys_init_data {
  dict begin
  /content dict def
  content begin
    /type  edef
    /value edef
    /length { value length } def
  end
  content
  end
} def
```

`sys_init_data` opens a new variable-scope by defining a new, anonymous dictionary and opening it. In this new scope, another newly created dictionary is defined by the name `content`. This content-dictionary receives three entries: `type`, `value` and the method `length`. Each entry value is taken from the top of GML's stack. The newly created dictionary is then pushed onto the stack and the current scope is destroyed by closing the current dictionary, leaving the anonymous dictionary on the stack. In GML notation, a JavaScript-variable's content is defined by pushing the actual value and a pre-defined constant to identify the type of the variable (such as `Types.number`, `Types.array`, etc.) onto the stack, and calling `sys_init_data`. The translator prefixes all JavaScript-identifiers with `usr_` (in order to ensure that all declarations of identifiers do not collide with predefined GML objects) and uses the following translations:

Undefined: Variables of type `undefined` result from operations that yield an undefined result or by declaring a variable without defining it. `var x;` leads to `x` being of type `undefined`. It is translated to

```
/usr_x Nulls.Types.undefined
       Types.undefined sys_init_data def
```

Boolean: In JavaScript, boolean values are denoted by the keywords `true` and `false`. The translation simply maps these values to equivalent numerical values in GML, which interprets them. The JavaScript-statement `var x = true;` becomes

```
/usr_foo 1 Types.bool sys_init_data def
```

Number: All JavaScript numbers (including integers) are represented as 32-bit floating point values. As GML stores numbers as 32-bit floats internally as well, we simply map them to GML's number representation. For the sake of completeness, `var x = 3.14159;` is translated to

```
/usr_x 3.14159 Types.number sys_init_data def
```

String: Although GML does support strings, they cannot be accessed character-wise. We cope with this limitation by defining strings as GML-arrays of numbers. Each number is the Unicode of the respective character. As GML allows to retrieve and to set array-elements based on indexes, this approach meets all conditions of JavaScript-strings. The statement `var x = "Hello World";` becomes

```
/usr_x
  [ 72 101 108 108 111 32 87 111 114 108 100 ]
  Types.string sys_init_data def
```

Array: JavaScript arrays allow to hold data with different types, the array's contents may be mixed. This behavior is in line with GML. The JavaScript-example `var x = [true, false, "maybe"];` has a straightforward translation:

```
/usr_x [ 1 Types.bool sys_init_data
         0 Types.bool sys_init_data
         [109 97 121 98 101]
         Types.string sys_init_data ]
         Types.array sys_init_data def
```

Object: In JavaScript an object consists of key-value-pairs, e.g., `var x = {x: 1.0, y: 2.0, z: 42};` This structure is mapped to nested GML-dictionaries. The value of a variable's content is a dictionary of its own. This dictionary contains the entries corresponding to JavaScript-object's members, which are also defined as variable contents.

The example above defines a JavaScript-object of name `x` with key-value-pairs `x` to be 1, `y` to be 2, and `z` to be 42:

```
/usr_x dict begin
  /obj dict def obj begin
   /usr_x  1.0 Types.number sys_init_data def
   /usr_y  2.0 Types.number sys_init_data def
   /usr_z 42.0 Types.number sys_init_data def
  end obj
Types.object sys_init_data end def
```

Opening an anonymous dictionary creates a new scope. In this scope, a dictionary is created and bound to the name `/obj`. It is then opened and its members are defined, just like anonymous variables would be. The object dictionary is then closed, put on the stack, and used to define an anonymous variable. The enclosing anonymous scoping dictionary is then closed and simply discarded.

JavaScript objects may hold functions. Our translator *Euclides* handles JavaScript object-functions like ordinary functors (next subsection) and assigns their internal name to a key-value-pair.

Function: JavaScript has first-class functions. Therefore, it is possible to assign functions to variables, which can be passed as parameters to other functions, for example. In the following example, a function `function do_nothing() {}` is declared and defined. Afterwards, the function is assigned to a variable `var x = do_nothing;`. If we abstract away from the translation of the function `do_nothing`, the statement `var x = do_nothing;` becomes:

```
/usr_do_nothing {
  %% ... definition of function omitted ...
} def

/usr_x
  /usr_do_nothing Types.function
sys_init_data def
```

In JavaScript, `x` can now be used as a functor, which acts the same ways as `do_nothing`. Because such functors can be reassigned, it is necessary to handle functor calls (`x()`) differently than ordinary function calls (`do_nothing()`). In this situation *Euclides* creates a temporary array, which contains the functor parameters and passes this array as well as the variable referencing the function name to a system function `sys_execute_var`. This system function resolves the functor and determines the referenced function, unwraps the array and performs the function call.

Functions: In GML, functions are defined using closures, such as `/my_add { add } def`. If this function `my_add` is executed, the closure `{ add }` is put onto the stack, its brackets are removed, and the content is executed.

To execute a GML function, its parameters need to be put on the stack prior to the function call: `1.0 2.0 my_add` The resulting number `3.0` will remain on the stack. Please note, that GML functions may produce more than one result (left on the stack) at each function call. This allows to define functions with more than one result value. Following JavaScript, called functions return only one value by convention. The number and names of function parameters are known at compile time. Only functors (referenced functions stored in variables) may change at run time and cannot be checked ahead of time.

Translated functions and parameters are named just like their JavaScript-counterparts (except for their `usr_` prefix).

Scopes: As JavaScript uses a scoping mechanism different to GML, it has to be emulated. This is a rather difficult task, which has to take the following properties of JavaScript scopes into account.

- JavaScript functions may call other functions or themselves.
- Called functions may declare the same identifiers as the calling functions.
- Within functions other functions may be defined.
- Blocks might be nested inside functions, redefining symbols or declaring symbols of the same name.

The translator uses GML's dictionary mechanism to emulate JavaScript-scopes. A dictionary on the dictionary stack can be opened and it will take all subsequent assignments to GML-identifier (variables). Since only the opened dictionary is affected, this behavior is the same as the opening and closing scopes in different scoped programming languages, such as C or Java.

Thus an assignment `/x 42 def` can be put into an isolated scope by creating a dictionary (`dict`), opening it (`begin`), performing the assignment, and closing the dictionary (`end`). The following example shows how such GML scopes can also be nested:

```
dict begin
  /x 3.141 def             %% x is 3.141
  dict begin               %%
    /x 4 def               %% x is 4.0
  end                      %% x is 3.141
end                        %% x is unknown
```

As noted before, JavaScript supports redefinition of identifiers that were declared in a scope below the current one. Fortunately, GML exhibits just the same behavior when reading out the values of variables/keys from dictionaries of the dictionary stack. Consequently, the following example works as expected.

```
dict begin
  /x 42 def
  dict begin
    /y x 1 add def         %% y is now 43
  end
end
```

However, assignments to variables have to be handled differently in GML. The Generative Modeling Language does not distinguish between declaration and definition, any declaration must be a definition and vice versa.

The translator solves this problem. It uses a system function called `sys_def`, which is included into all translated JavaScript sources automatically. This function applies GML's `where` operator to the dictionary stack in order to find the uppermost dictionary, where the searched name is defined. The operator returns the reference to the dictionary, in which the name was found.

Control Flow for Functions: The Generative Modeling Language and all PostScript dialects lack a dedicated jump operation in control flow. Imperative functions often require the execution context to jump to a different point in the program at any time - and to return from there as well.

Fortunately, GML provides an exception mechanism. A GML exception is propagated down GML's internal execution stack until a `catch` instruction is encountered. In this way it overrides any other control structure it encounters. We use GML's exception mechanism to jump outside a function as illustrated in the following empty function skeleton:

```
/usr_foo {
  dict begin
  /return_issued 0 def
  { dict begin
    %% ... function body omitted ...
    end }
  { /return_issued 1 def }
  catch

  return_issued not
  { Nulls.Types.undefined
    Types.undefined sys_init_data } if
  end
  sys_exception_return_handler
} def
```

In this empty skeleton, the function opens a new anonymous scope. Inside this scope `dict begin ... end` the local identifier `/return_issued` is set to `0`. Afterwards a GML try-catch-statement `{ try_block } { catch_block } catch` contains the JavaScript-function implementation. In this translation, the catch block redefines `/return_issued` to `1` to indicate that a JavaScript `return` statement has been executed in the function body. JavaScript functions without any `return` statement automatically return `null` resp. in GML `Nulls.Types.undefined Types.undefined sys_init_data`. A corresponding JavaScript-return statement, e.g., `return 42;`, is translated to

```
42.0 Types.number sys_init_data end throw
```

In this example, the number `42.0` is put onto the stack. The actual function body's scope is closed `end`, and the `throw` operator is applied. The distinction of whether the end of the function body was reached by normal program flow or via a return statement determines, if a return value needs to be constructed (`null`) and put onto the stack.

Parameters to functions are simply put on the stack. The function body retrieves the expected number of parameters and assigns them to dictionary entries of the outer scope defined in the function translation. A complete example of a translated JavaScript-function shows the interplay of all mechanisms. The simple JavaScript-function

```
function foo(n) { return n; }
```

is translated to

```
/usr_foo {
  dict begin
  /usr_n edef
  /return_issued 0 def
  { dict begin
    usr_n
    end
    throw
    end }
  { /return_issued 1 def }
  catch

  return_issued not
  { Nulls.Types.undefined
    Types.undefined sys_init_data } if
  end
  sys_exception_return_handler
} def
```

A function call, for example `foo(3)`, yields the translation `3.0 Types.number sys_init_data usr_foo`. If we assign the function `foo` to a variable `foo_functor`, the calling convention in GML would change significantly.

```
/usr_foo_functor
  /usr_foo Types.function sys_init_data def
```

is called via

```
[ 3.0 Types.number sys_init_data ]
usr_foo_functor sys_execute_var
```

and represents the JavaScript call `foo_functor(3.0);`

Exceptions: The language JavaScript offers support for throwing exceptions as shown in the following example:

```
throw "Error: unable to read file.";
```

Its syntax is similar to a return statement. To implement such behavior, we also use GML's exception handling mechanism. The *Euclides* translator adds a call to the predefined system function `sys_exception_return_handler` at the end of each translated function (see example above).

Throwing an exception in JavaScript translates into a global GML variable `exception_thrown` being set to 1, closing the current dictionary and calling GML's `throw`. The `sys_exception_return_handler` will check if an actual exception is being thrown, and if so, calls `throw` again. A catch-block inside a JavaScript program would set `exception_thrown` to 0.

Operators: The evaluation of expressions demands variables to be accessed. While GML provides operators that operate on their own set of types, they obviously cannot be used to access the translated/emulated JavaScript-variables. For this reason, the *Euclides* translator automatically includes a set of predefined GML functions that substitute operators defined in JavaScript.

Value Access: Performing the opposite operation to `sys_init_data`, `sys_get_value` will retrieve the data saved in a JavaScript-variable resp. its GML-dictionary. For example, to retrieve `v.value` the function `sys_get_value` is applied to `v`.

```
/sys_get_value { begin value end } def
```

Element Access: The system function `sys_get` implements string, array and object access. Applied to a string / an array `Arr` and index `k`, it will return the element `Arr[k]`. If its parameters are an object `Obj` and an attribute `name`, the function `sys_get` executes `Obj.name`. This may result in a value, which is put on the stack or in a function, which is called. Conforming to JavaScript, it returns JavaScript `undefined` for any requested elements that do not exist.

```
/sys_get {
  dict begin
  /idx exch def /var exch def

  var.type Types.string eq {
    %% ... handling strings ...
  } if

  var.type Types.array  eq {
    %% ... handling arrays  ...
  } if

  var.type Types.object  eq {
    var sys_get_value idx known 0 eq {
      %% return null, if element
       %% does not exist
       Nulls.Types.undefined
       Types.undefined sys_init_data
    } if
    var sys_get_value idx known 0 ne {
      %% access element
      var sys_get_value idx get
    } if
  } if
  end
} def
```

Analogous to `sys_get`, `sys_put` inserts data into strings and arrays, or defines members of objects. If `sys_put` encounters an index k that is out of an array's range, the array is resized and filled with JavaScript `undefined`s.

Functors: The already mentioned routine `sys_execute_var` inspects a given variable. If it is a function, it will retrieve the array supplied to hold all parameters and execute the function. The dynamic binding of functions to variables requires to consider two situations at run time: The functor receives the correct amount of parameters for its function, or the number of parameters does not correspond to the referenced function. In the latter case, the function is not called and `null` is returned instead.

At compile time, a function is defined to expect a concrete number of parameters. This information is kept to perform parameter checks at run time. In this way, the correct number of parameters for all functors can be determined any time.

JavaScript built-in Operators: To illustrate the translation of relational, arithmetical or bit-shift operators defined by JavaScript, we discuss the equal operator ==. It is (like all such operators) mapped to a corresponding routine `sys_eq`. Depending of the operands' types it delegates the comparison to subroutines such as `bool_eq`, `string_eq` or `array_eq` that perform the actual comparison. If the types and the values do match, `sys_eq` directly returns the JavaScript-value `true`. If types do not match, the variable is converted to the type of the respective operand, as specified by JavaScript, and then compared.

Control Flow: The JavaScript if-then-else statement corresponds one-to-one to the same GML statement. Consequently, the conditional expression is translated straightforwardly. Using the expression mapping introduced in the previous section (e.g. `sys_eq` implements the equality operator), the JavaScript statement

```
if(a == b) { c = a; } else { c = b; }
```

is translated into:

```
%% if (a==b)
usr_a usr_b sys_eq sys_get_value
{ %% then:
  dict begin {
    dict begin
      /usr_c usr_a sys_def
    end
  } exec end
}
{ %% else:
  dict begin {
    dict begin
      /usr_c usr_b sys_def
    end
  } exec end
} ifelse
```

The exec-statements (and their closures) stem from the fact that both sub-statements, the then-part and the else-part, are statement blocks { ... }. These blocks are executed within their own, new scopes.

Loops: GML supports different types of looping control structures, which have similar names to JavaScript-loops (e.g., both languages have a for-loop). However, the GML counterparts have different semantics (e.g., GML's for-loop has a fixed, finite number of iterations, which is known before execution of the loop body, whereas JavaScript-loops evaluate the stop condition during execution, which may result in endless loops). The *Euclides* translator uses the GML `loop` mechanism, which is an infinite loop that can be quit using the `exit` operator.

An important problem is that control structures such as for, while and do-while are not only controlled by the loop's stop condition, but also by JavaScript statements such as `continue` and `break` within the loop body (besides `return` and `throw` as mentioned before). The statement `break` immediately stops execution of the loop and leaves it, whereas `continue` terminates the execution of the current loop iteration and continues with the next iteration of the loop. Therefore, we translate an empty while loop `while(false) { ... }` to

```
{  /continue_called 0 def
  {  0 Types.bool sys_init_data
    sys_get_value not { exit } if
    {  dict begin
      %% ... loop body omitted ...
      end
    } exec
  } loop
  continue_called not { exit } if
} loop
```

GML's `exit` keyword terminates the current loop. This behavior is leveraged by the *Euclides* translator to implement `break` and `continue`. The translation uses two nested loops that will run infinitely.

Prior to the begin of the inner loop `/continue_called` is set to `0`. At the top of the inner loop, the loop condition is tested. If the condition evaluates to `false`, the inner loop is exited using GML's `exit`. Otherwise a new scope is created and the loop-statement executed within that scope.

During loop iterations, there are three scenarios under which a loop can terminate:

1) If the loop condition is met: When the condition evaluates to `false`, the inner loop is exited. Since `continue_called` is not set to `true`, the outer loop will terminate as well.
2) If the loop body encounters JavaScript `break` (resp. GML `exit`): Again, the inner loop is left. `continue_called` will not be set to `true`, hence the outer loop will also terminate.
3) If the function returns: GML's exception throwing mechanism will unwind the stack until the catch-handler at the end of the function is encountered.

If the loop body encounters a JavaScript-`continue` statement, `continue_called` will be set to `true` and the GML `exit` command will immediately stop the inner loop. Since `continue_called` is set, execution does not leave the outer loop, however. As a consequence, `continue_called` becomes `0` again, and execution re-enters the inner infinite loop.

The do-while-statement is translated very similar to the while-statement. The only semantic differences in JavaScript are that execution will enter the loop regardless of the loop-condition and that the loop-condition is tested after loop body execution. *Euclides* translates an empty do-while-statement `do { ... } while(false)` as follows:

```
{  /continue_called 0 def
  { {  dict begin
      %% ... loop body omitted ...
      end
    } exec
    0 Types.bool sys_init_data
      sys_get_value not { exit } if
  } loop
  continue_called not { exit } if
  0 Types.bool sys_init_data
  pop
} loop
```

Due to a semantic difference of JavaScript `continue` in do-while-loops, this statement needs to be handled differently. If `continue` is encountered, the loop condition must still execute before the loop body is re-entered, because side effects inside the loop condition may occur (such as incrementing a counter). *Euclides* handles this problem by executing the condition expression a second time in the outer loop. Since expressions always return values, any value resulting from the loop-expression has to be popped off the stack.

Although GML has a `for` operator, it is semantically incompatible with JavaScript's one. Its increment is a constant number, and so is the limit. In JavaScript, both increment and limit must be evaluated at each loop body execution. Therefore, we translate `for` just like the previous constructs by two nested loops with the increment condition repeated in outer loop (due to `continue` semantics). Finally, *Euclides* translates the JavaScript statement `for (var i=0; i<1; i++) { }` to GML via

```
dict begin
%% initialization (i=0)
/usr_i 0.0 Types.number sys_init_data def
{  /continue_called 0 def
  {  %% condition (i<1)
    usr_i 1.0 Types.number
    sys_init_data sys_lt
    sys_get_value not { exit } if
    {  dict begin
      %% ... loop body ...
      end
    } exec
    %% increment (i++)
    usr_i
      usr_i 1 Types.number
      sys_init_data sys_add
    /usr_i sys_edef
    pop
  } loop
  continue_called not { exit } if
  %% increment again (i++)
  usr_i
    usr_i 1 Types.number
    sys_init_data sys_add
  /usr_i sys_edef
  pop
} loop
end
```

In JavaScript, the following for-in statement `for(var x in array) statement;` is semantically equivalent to:

```
for(var i = 0; i < array.length; i++) {
  var x=array[i]; statement;
}
```

This construction loops over the elements of an array provides the loop body with a variable holding the current element.

Selection Control Statement: The translation of the JavaScript `switch` statement poses several difficulties:

- If a case condition is met, execution can "fall through" till the next `break` is encountered.
- If a `break` is encountered, the currently executed `switch` statement must be terminated.
- Of course, `switch` statements may be nested.

To develop a semantically consistent solution, we did not want to alter the translation of JavaScript-`break` inside switch statements (compared to loops). We solve the problem of breaking outside the `switch` statement by implementing it as a loop that is run exactly once. In GML it reads like `1 {` *`loop_instructions`* `} repeat`. This way our translation of `break` shows semantically correct behavior, it terminates the loop. Consider the following JavaScript-program:

```
var x = 0, y = 0;

function bar() { return 3; }

function foo(i) {
  switch(i) {
    case 0:
    case 1:
    case 2: x = 1;
    case 4: x = 3;
    case bar(): x = 2; break;
    default: y = 5;
  }
}
```

The function `foo` will be translated to:

```
/usr_foo
{ dict begin
  /usr_i edef
  /return_issued 0 def
  { dict begin
    /switch_cnd_met1 0 def
    1 { usr_i 0.0
      Types.number sys_init_data
      sys_eq sys_getvalue
      switch_cnd_met1 1 eq or {
        /switch_cnd_met1 1 def
      } if

      usr_i
      1.0 Types.number sys_init_data
      sys_eq sys_getvalue
      switch_cnd_met1 1 eq or {
        /switch_cnd_met1 1 def
      } if

      usr_i
      2.0 Types.number sys_init_data
      sys_eq sys_getvalue
      switch_cnd_met1 1 eq or {
        /switch_cnd_met1 1 def
        %% x = 1;
        /usr_x 1.0 Types.number
          sys_init_data sys_def
      } if

      usr_i
      4.0 Types.number sys_init_data
      sys_eq sys_getvalue
      switch_cnd_met1 1 eq or {
        /switch_cnd_met1 1 def
        %% x = 3;
        /usr_x 3.0 Types.number
        sys_init_data sys_def
      } if

      usr_i usr_bar
      sys_eq sys_getvalue
      switch_cnd_met1 1 eq or {
        /switch_cnd_met1 1 def
        %% x = 2;
        /usr_x 2.0 Types.number
        sys_init_data sys_def
        exit
      } if
      %% y = 5;
      /usr_y 5.0 Types.number
      sys_init_data sys_def
    } repeat
    currentdict /switch_cnd_met1 undef end
  }
  { /return_issued 1 def } catch

  return_issued not {
    Nulls.Types.undefined
    Types.undefined sys_init_data
  } if
  end
  sys_exception_return_handler
} def
```

This example shows that we introduce an internal variable `/switch_cnd_metX` for traversing the case statements. As soon as a case statement condition is met, `/switch_cnd_metX` is set to `true`, leading execution into every encountered case statement.

The *Euclides* translator takes into account that switch statements may be nested. As it traverses the AST, it keeps book of all internal variable to ensure a unique name (`switch_cnd_met1`, `switch_cnd_met2`, ..., `switch_cnd_metN`).

The example translation shows that for `foo(3)` the cases 0, 1, 2, 4 and 3 (= `bar()`) will only execute case 3, where the `1 { } repeat` statement will be broken out of with the GML `exit` operator. The default block will be executed in any case if execution is still inside the `repeat` statement, no further state is checked for `default`.

The JavaScript to PostScript translation target reduces the inhibition threshold of the GML significantly. Even advanced GML users, who already know how to program in PostScript style, can use Euclides to translate algorithms, which are often presented in a imperative, procedural (pseudo-code) style [38].

V. Conclusion and Future Work

The correct translation of control flow structures to various target platforms is a non-trivial task. For example, due to the fact that there is no concept of goto in the PostScript language and its dialects, the main challenge is the complete translation of JavaScript into a PostScript dialect including all control flow statements. To the best of our knowledge, this is the first complete translator. Other projects (PdB by Arthur van Hoff, pas2ps by Dulith Herath and Dirk Jagdmann) do not support e.g., return statements.

The main contribution is the meta-modeler concept, which allows a user to export generative models to other platforms without losing its main feature the procedural paradigm. It is well suited for procedural modeling, has a beginnerfriendly syntax and is able to generate and export procedural code for various, different generative modeling or rendering engines. The source code does not need to be interpreted or unfolded, it is translated. Therefore, it can still be a very compact representation of a complex model.

The target audience of this approach consists of beginners and intermediate learners of procedural modeling techniques and addresses application domain experts (e.g., archaeologists in a cultural heritage project) who are seldom computer scientists. These experts are needed to tap the full potential of generative techniques.

The current IDE only offers basic functionality and some convenience when creating, editing or translating source code. For further improvements, we envisage using a Swing-based IDE framework like the NetBeans Platform, which offers a modular approach for creating rich applications. In the backend, further optimizations concerning the performance of the generated code are planned - e.g., more direct mapping onto native data types. Additional target languages would extend the application field of the framework.

The Euclides modeler is available in version 2.0 and can be downloaded at: http://www.cgv.tugraz.at/euclides.

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 (www.3Dcoform.eu) under grant number FP7 ICT 231809, from the Austrian Research Promotion Agency (FFG) for the research project METADESIGNER, grant number 820925/18236, as well as from the German Research Foundation (DFG) for the research project PROBADO under grant INST 9055/1-1 (www.probado.de).

References

[1] 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.

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

[3] 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.

[4] 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.

[5] 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.

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

[7] R. B. OpenGL Architecture, *OpenGL Reference Manual*, R. B. OpenGL Architecture, Ed. Addison-Wesley Publishing Company, 1993.

[8] NVidia, "NVIDIA CUDA C Programming Guide."

[9] D. Reiners, G. Voss, and J. Behr, "OpenSG: Basic concepts," *Proceedings of OpenSG Symposium 2002*, vol. 1, pp. 1–7, 2002.

[10] G. Voß, J. Behr, D. Reiners, and M. Roth, "A multi-thread safe foundation for scene graphs and its extension to clusters," *Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization*, vol. 4, pp. 33–37, 2002.

[11] B. Eckel, *Thinking in C++: Introduction to Standard C++, Practical Programming*, B. Eckel, Ed. Prentice Hall, 2003.

[12] 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.

[13] R. Berndt, G. Buchgraber, S. Havemann, V. Settgast, and D. W. Fellner, "A publishing workflow for cultural heritage artifacts from 3d-reconstruction to internet presentation," in *Digital Heritage. Third International Conference, EuroMed 2010*, M. Ioannides, D. W. Fellner, A. Georgopoulos, and D. Hadjimitsis, Eds., vol. 6436. Springer, 2010, pp. 166–178, doi:10.1007/978-3-642-16873-413.

[14] M. Strobl, R. Berndt, V. Settgast, S. Havemann, and D. W. Fellner, "Publishing 3D Content as PDF in Cultural Heritage," *Proceedings of the 10th International Symposium on Virtual Reality, Archaeology and Intelligent Cultural Heritage (VAST)*, vol. 6, pp. 117–124, 2009.

[15] V. Settgast, T. Ullrich, and D. W. Fellner, "Information Technology for Cultural Heritage," *IEEE Potentials*, vol. 26, no. 4, pp. 38–43, 2007.

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

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

[18] 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.

[19] P. Müller, G. Zeng, P. Wonka, and L. Van Gool, "Image-based Procedural Modeling of Facades," *ACM Transactions on Graphics*, vol. 28, no. 3, pp. 1–9, 2007.

[20] M. Lipp, P. Wonka, and M. Wimmer, "Interactive Visual Editing of Grammars for Procedural Architecture," *ACM Transactions on Graphics*, vol. 27, no. 3, pp. 1–10, 2008.

[21] P. Müller, P. Wonka, S. Haegler, U. Andreas, and L. Van Gool, "Procedural Modeling of Buildings," *Proceedings of 2006 ACM Siggraph*, vol. 25, no. 3, pp. 614–623, 2006.

[22] B. Lintermann and O. Deussen, "A Modelling Method and User Interface for Creating Plants," *Computer Graphics Forum*, vol. 17, no. 1, pp. 73–82, 1998.

[23] B. Ganster and R. Klein, "An Integrated Framework for Procedural Modeling," *Proceedings of Spring Conference on Computer Graphics 2007 (SCCG 2007)*, vol. 23, pp. 150–157, 2007.

[24] D. Finkenzeller, "Detailed Building Facades," *IEEE Computer Graphics and Applications*, vol. 28, no. 3, pp. 58–66, 2008.

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

[26] E. Mendez, G. Schall, S. Havemann, D. W. Fellner, D. Schmalstieg, and S. Junghanns, "Generating Semantic 3D Models of Underground Infrastructure," *IEEE Computer Graphics and Applications*, vol. 28, pp. 48–57, 2008.

[27] C. Reas and B. Fry, *Processing: A Programming Handbook for Visual Designers and Artists*. MIT Press, 2007.

[28] S. Davidson, "Grasshopper- generative modeling for rhino," online: http://www.grasshopper3d.com/, 2011.

[29] D. Flanagan, *JavaScript. The Definitive Guide*, 5th ed. O'Reilly Media, 2006.

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

[31] T. Ullrich, U. Krispel, and D. W. Fellner, "Compilation of Procedural Models," *Proceeding of the 13th International Conference on 3D Web Technology*, vol. 13, pp. 75–81, 2008.

[32] C. Schinko, T. Ullrich, T. Schiffer, and D. W. Fellner, "Variance Analysis and Comparison in Computer-Aided Design," *Proceedings of the International Workshop on 3D Virtual Reconstruction and Visualization of Complex Architectures*, vol. XXXVIII-5/W16, pp. 3B21–25, 2011.

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

[34] R. Hammer, M. Hocks, U. Kulisch, and D. Ratz, *C++ Toolbox for Verified Computing*, R. Hammer, M. Hocks, U. Kulisch, and D. Ratz, Eds. Springer, 1997.

[35] A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, A. Griewank and A. Walther, Eds. SIAM, 2008.

[36] M. L. Keler, "On the Theory of Screws and the Dual Method," *Proceedings of A Symposium Commemorating the Legacy, Works, and Life of Sir Robert Stawell Ball Upon the 100th Anniversary of "A Treatise on the Theory of Screws"*, vol. 1, pp. 1–12, 2000.

[37] S. Havemann and D. W. Fellner, "Generative Parametric Design of Gothic Window Tracery," *Proceedings of the 5th International Symposium on Virtual Reality, Archeology, and Cultural Heritage*, vol. 1, pp. 193–201, 2004.

[38] T. H. Cormen, C. Stein, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, T. H. Cormen, C. Stein, C. E. Leiserson, and R. L. Rivest, Eds. B&T, 2001.