
GCL Viewer

A study in improving the understanding of GCL programs

- Master thesis -
By Ibrahim Bokharouss

Supervisors:

Dr. M.R.V. Chaudron (TU/e)
Ziad Bizri (Google)

Abstract

Reading and understanding source code is an essential activity of software development. Fixing bugs, adding features or making changes to a software system requires the examination and comprehension of the related source code. Software Visualization is a field of study that uses visual techniques (among others) with the aim to enhance the understanding of software.

In this thesis we study how we can improve the understanding of programs written in the General Configuration Language (GCL), a programming language developed at Google Inc. We present GCL Viewer, a software visualization tool that we have designed to improve the understanding of GCL programs. We have designed GCL Viewer as a Rich Internet Application, allowing us to benefit from the advantages of web applications (e.g. low deployment cost, portability, integration with web services) while maintaining the responsive and interactive aspects that we know from desktop applications.

To evaluate our solution we have conducted a user evaluation in the form of a survey. The survey was designed to measure to what extent GCL Viewer helps in working with GCL programs, and to determine which features of the tool were perceived as being most successful in achieving this. The survey results show that GCL Viewer significantly helps in understanding GCL programs. The features that were designed to provide an explicit view on a program (e.g. see the evaluated values of expressions) were rated best.

Acknowledgements

Although writing a thesis seems like highly individual work, I could have never managed to accomplish this without the help of the people around me.

First, I would like to thank my supervisors Dr. Michel Chaudron and Ziad Bizri for the great cooperation and for teaching me a variety of useful techniques, approaches and insights. They were both great advisors and mentors to me, and I'm very grateful for the opportunity to have worked with them. I would like to thank Dr. Michel Chaudron also for his time and support outside my thesis work. He always invested time and showed interest whenever I felt the need to discuss something with him.

Further I would like to thank prof. Jack van Wijk for his support and useful contributions to my work, and for inspiring me on the world of (software) visualization. I also thank the GCL team, the UX team, and all the people at Google that provided useful input to my work. Furthermore, I would like to thank all the participants in the survey study.

I thank my parents and my friends, notably, Aga Matysiak, Ehsan Baha, Jennifer Kerkvliet, Lisa Ableitinger, Mike Holenderski and for they were a very important support to me throughout the writing of this thesis.

Last but certainly not least, I would like to thank the members of my exam committee, Prof. Jack van Wijk, Dr. Michel Chaudron, and Prof. Mark van den Brand.

Ibrahim Bokharouss

Table of contents

1. INTRODUCTION	10
1.1 GENERAL INTRODUCTION	10
1.2 THESIS PROJECT	10
1.3 APPROACH	11
1.4 A BRIEF INTRODUCTION TO GCL VIEWER	11
1.5 THESIS OUTLINE	11
2. INTRODUCTION TO GCL	12
2.1 GCL	12
2.2 MAIN CONCEPTS	12
2.2.1 Values	12
2.2.2 Expressions	13
2.2.3 Functions	13
2.2.4 Tuples	13
2.2.5 Tuple composition	14
2.2.6 Tuple inheritance	15
2.2.7 Expansions and parameterized tuples	15
2.2.6 References	16
2.2.7 Imports	18
2.2.8 Templates	18
2.2.9 Assertions	19
2.2.10 Objects: Typed tuples	20
2.3	20
3. ANALYSIS AND MOTIVATION	22
3.1 HISTORICAL OUTLINE	22
3.1.1 Evolution of complexity	22
3.1.2 Agile software development process	23
3.1.3 A growing need for tooling	23
3.2. KEY ISSUES	23
3.2.1 Complex expressions	23
3.2.2 Unclear definition of a configuration	24
3.2.3 Unknown origin of a variable	25
3.2.4 Hidden Dependencies	27
3.2.5 Unintended modifications	27
3.2.7 Cascaded inheritance	28
3.2.7 Lack of overview in a configuration	29
3.2.8 Erroneously defined parameterized tuples	30
3.2.9 Ambiguous references	31
4. TOOL DESIGN	31
4.1 GCL VIEWER	32
4.2 DESIGN PROCESS	34
4.3 METHODS AND TECHNIQUES	34
4.3.1 A visual tool	34
4.3.2 Visual techniques	34
4.3.3 Hypertext browsing	36
4.3.4 Hybrid interaction control	36
4.3.5 Color techniques	36
4.3.6 Font techniques	37
4.4 MAIN FEATURES	37
4.4.1 Evaluate expression	37
4.4.2 The Expanded View	38
4.4.3.1 The origin of a field	39
4.4.3.2 Locating the origin of a field	39
4.4.4.1 Dependency Listing	39
4.4.4.2 Dependency Coloring	40
4.4.5 Modification tags	40

4.4.6 Inheritance info	41
4.4.7.1 Skeleton View.....	43
4.2.7.2 View Key tuples	44
4.4.8 Browsing through a parameterized tuple	45
4.4.9 Navigating references	46
4.5 ADDITIONAL FEATURES.....	46
4.5.1. Enhancing the readability of configurations.....	46
4.5.2 Reporting evaluation errors and warnings.....	48
4.5.3 Instant assertion result.....	49
4.5.4 Field inspector.....	49
4.5.5 Hybrid control	50
4.6 USABILITY	51
4.7 SUMMARY.....	51
5. TECHNICAL DESIGN	53
5.1 PRELIMINARIES	53
5.1.1 Design goals	53
5.1.2 Processing a GCL file.....	54
5.2 DESIGN DECISIONS	56
5.2.1 Distributed computing.....	56
5.2.2 Web application.....	57
5.2.3 Rich Internet Application.....	58
5.3 ARCHITECTURAL DESIGN	60
5.3.1 System architecture	60
5.3.2 Physical view.....	62
5.3.3 Data interchange format.....	63
5.3.4 Operational view	64
5.3.5 Fault tolerance	66
5.4 BACKEND SERVICE	66
5.4.1 Purpose	66
5.4.2 Method and techniques	66
5.4.3 Detailed design.....	67
5.5 FRONTEND SERVICE	69
5.5.1 Purpose	69
5.5.2 Method and techniques	69
5.5.3 Detailed design.....	70
5.6 CLIENT APPLICATION	72
5.6.1 Purpose	72
5.6.2 Method and techniques	72
5.6.3 Detailed design.....	77
5.7 SUMMARY.....	79
6. EVALUATION	80
6.1 EVALUATION GOALS	80
6.2 SURVEY DESIGN	80
6.3 SET UP.....	81
6.4 RESULTS AND ANALYSIS.....	81
6.4.1 Results for all participants.....	81
6.4.2 Results grouped by experience with GCL Viewer.....	85
6.4.3 Results grouped by experience with configuration languages.....	87
7. RELATED WORK.....	90
7.1 HISTORY	90
7.2 TERMINOLOGY	90
7.3 CLASSIFYING GCL VIEWER	90
7.4. RELATED TOOLS	91
7.4.1 Pretty printing.....	92
7.4.2 SID: A graphical browser.....	92
7.4.3 Enhancing Code for Readability and Comprehension Using SGML	92
7.4.4 CodeSurfer.....	92
7.4.5 Fisheye View.....	93
7.5 OTHER SOFTWARE VISUALIZATION TOOLS.....	93
7.5.1 Graph based software visualization tools.....	93

7.5.2 <i>Visualization of software evolution</i>	93
7.6 OTHER RELATED WORK	94
7.6.1 <i>Embedding visualization into electronic documents and reports</i>	94
7.6.2 <i>Using the Web as a platform for visualization tools</i>	94
8. CONCLUSIONS	95
9. FUTURE WORK	97
10. PROJECT EVALUATION	98
REFERENCES	99
APPENDIX A. [REDACTED]	102
APPENDIX B. USAGE STATISTICS FOR GCL VIEWER	103
APPENDIX C. FEEDBACK FROM GCL VIEWER USERS	105
APPENDIX D. SCREENSHOTS	106

1. Introduction

1.1 General introduction

In the software industry engineers spend a great deal of their time on reading and understanding source code. An activity where the understanding of source code plays a key role is software maintenance. Industry practice studies have shown that software maintainers spend 50% of their time on understanding of code [Standish84]. Code reuse is another activity in which understanding of source code plays a key role. Practices such as component based software engineering and object oriented programming have given a steep rise to the reuse of code in the last decades. Code reviewing, which is a common quality assurance practice in industry, is also an activity where the understanding of source code is essential.

The ability of people to understand programs is directly related to the ease with which the source code and documentation can be read [Mayrh95]. But in general, software has increased in size and complexity over time, leading to code that is harder to read and understand. Software systems spanning millions of lines of codes are not uncommon these days, while the increased share and reuse of code has led to an explosion of dependencies within the source code. The ever increasing complexity of software fueled the need to study how we can improve the readability and comprehensibility of software. Software visualization is a field of study that has emerged out of this. Software Visualization is defined as the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction technology to facilitate both the human understanding and effective use of computer software [Price93]. The indirect objectives of software visualization are to simplify the work on software (e.g. maintenance, code reviews), and reduce the time that an engineer spends on these activities.

1.2 Thesis project

In this thesis project we studied how software visualization can help the understanding of programs in a large scale industry setting. We chose the Google company as the setting of our study. The amount of data processed daily at Google and the complex software that steers this are a unique phenomena in the software industry, providing us with an interesting opportunity to study the benefits of software visualization in a large scale setting.

Our study focuses on a specific programming language that has been developed at Google: the General Configuration Language (GCL). GCL is used for configuring software systems. [REDACTED] Since its introduction in 2004, the complexity of both GCL and GCL programs have increased radically. The resulting code base has become hard to understand and maintain, as have been repeatedly reported by GCL users.

The primary goal of the project is to design a tool that improves the understanding of GCL and GCL programs, and consequently, a tool that helps the users to maintain, debug, and validate their GCL programs.

The secondary goals of the project are:

- A. Document how software visualization can help with the understanding of software in a large scale industry setting.
- B. Evaluate our solution to determine which aspects of the tool proved to be most successful and use this to make recommendations for software visualization for other languages.

- C. Advocate visualization tools implemented as Rich Internet Applications [Bozzon06, Yu06, Loosley06, Paulson05], which can be seamlessly integrated with online resources (e.g. software documentation, tutorials). The objective of the integration is to assist in the understanding of the presented material, and to add interactivity to the material.

1.3 Approach

As a first step of our study we have conducted an analysis of GCL and GCL programs with the objective to capture the key issues that contribute to the difficult understanding of GCL (programs). The discussions with and feedback from the core GCL user group were a vital input to this analysis.

In the next step we have designed and implemented a set of (visual) techniques that address these issues and aim to improve the understanding of GCL programs. This has resulted in our software visualization tool: GCL Viewer. The tool also provides a means of analyzing and validating GCL programs.

Finally, we have conducted a user evaluation to evaluate our solution. The user evaluation was conducted in the form of a survey, which we carried out 7 months after the tool had been deployed. With the survey we measured to what extent the tool accomplished the goal of helping GCL users in their work with GCL (programs). The survey also evaluates the usefulness of the features of the tool, with the objective to determine which aspects of the tool were perceived as being most useful for understanding GCL programs.

1.4 A brief introduction to GCL Viewer

GCL Viewer is a software visualization tool on a source code level. It provides an enriched view on GCL programs with the goal to improve the understanding of these programs. It also provides a means to analyze and validate GCL programs.

We have implemented GCL Viewer as a Rich Internet Application [Bozzon06, Yu06, Loosley06, Paulson05]. GCL Viewer renders its user interface as a HTML document and uses JavaScript to provide interaction. It exploits the client-rendering technique and uses a novice template system for rendering. Its architecture has been designed to support the reuse of the tool for languages derived from GCL but also other languages that have a similar structure to GCL.


The web application approach allows us to integrate the tool into web documents, which provides some additional useful applications of the tool, e.g. integration with online documentation (wikis, tutorials) to offer rich and interactive material.

1.5 Thesis Outline

The outline of this thesis is as follows. In chapter 2 we provide an introduction to the GCL language. In chapter 3 we present an analysis of the complexity of GCL (programs). In chapter 4 we present our tool, GCL Viewer, and discuss the visual techniques that we have applied. In chapter 5 we describe the technical aspects of the tool. The architecture of the system and the design of its components will be described in here. In chapter 6 we present the results of our user evaluation. In chapter 7 we describe the related work. We conclude the thesis with our conclusions (chapter 8), future work (chapter 9), and a project evaluation (chapter 10).

2. Introduction to GCL

In this chapter we give an informal introduction to the General Configuration Language (GCL). We describe the concepts of the language that are relevant to this thesis, and illustrate each of them with examples (section 2.2).

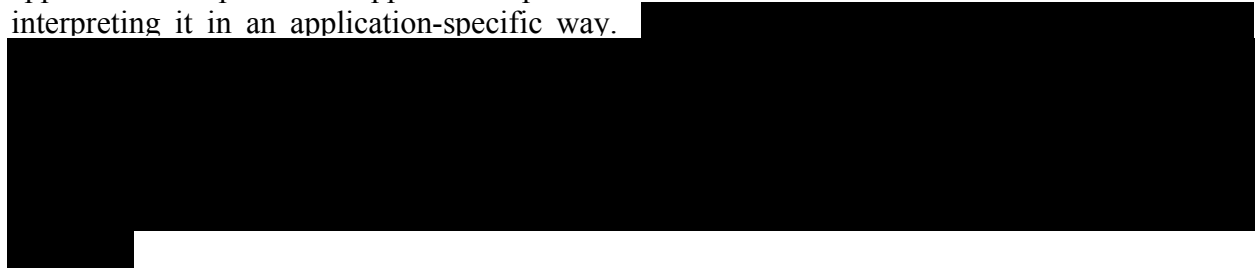


2.1 GCL

GCL is a language that is primarily used for specifying configurations. A configuration defines a set of attribute-value pairs (also denoted as *parameters*). GCL has a declarative nature. It is used to describe *what* a configuration should be like, without specifying *how* this going to be realized. Moreover, GCL files are statically evaluated; no run-time execution is required. The result of evaluating a GCL file is a hierarchical structure of attribute-value pairs. The resulting structure is fed into an application specific system, which interprets the configuration and ensures that the configuration is executed.

In essence a GCL configuration defines a hierarchical structure of attribute-value pairs. The power of GCL is that it offers constructs to describe these attribute-value pair structures in a convenient and efficient way, e.g. expressions, functions, inheritance. Another characteristic of GCL is that it allows you to specify sandboxed configurations; GCL configurations are not allowed to read arbitrary files or run arbitrary code.

GCL is designed to be a *framework* for specialized configuration languages. GCL is therefore application-independent. Application-specific meaning is given to a particular configuration by interpreting it in an application-specific way.



2.2 Main concepts

2.2.1 Values

A GCL configuration specifies a collection of values. Values come in various types and include the Boolean truth values `true` and `false`, 64bit integers, 64bit floating-point numbers, Unicode strings, composite values such as lists and maps, the special symbols `external` and `null`, and *tuples* (see section 2.2.4). The types need not to be declared explicitly, but are assigned implicitly. Below are some examples of values in GCL.

```

// a boolean value
true

// three integers
-42      // = -42
0x7f     // = 127
256M     // = 256 * 1048576 (the M stands for Mega)

// pi
3.14159265

// a string (the standard C escape sequences are supported)
"foobar\n"

// a list of the values 2, 7, 8, 9, 10, 12, and 20
[2, 7 .. 10, 12, 20]

// a map mapping weekdays to numbers
["su" : 0, "mo" : 1, "tu" : 2, "we" : 3, "th" : 4, "fr" : 5, "sa" : 6]

```

2.2.2 Expressions

Expressions can be used to compute new values by applying operators to other values (operands). GCL supports the usual arithmetic, string, comparison, and logical operations. Here are some examples of GCL expressions:

```

(1 + 1.5) * 3      // = 7.5
"foo" + "bar"     // = "foobar"
"hello" >= "foo"  // = true
true && false     // = false

```

2.2.3 Functions

Functions can be used as part of an expression. GCL offers a set of built in functions, and also allows self defined Lambda functions.

```

// Built in functions

substr("foobar", 3, 3) // = "bar"
tail([ 1, 2, 3, 4 ], 2) // = [ 3, 4 ]
cond(2 < 3, 4, 5)     // = 4

// Lambda functions

foo = lambda x: x + 5
bar = 2 * foo(3)      // = 16

```

2.2.4 Tuples

Tuples are sets of attribute-value pairs. Attributes are names (identifiers) that identify a particular value. An attribute-value pair is also called a *field*. The range of visibility of an attribute is called its *scope*. Each tuple automatically defines a new scope starting with the

opening brace and ending with the closing brace. All attributes within the same scope must have different identifiers. A field can also have a tuple as a value. We refer to this as a nested tuple. Nested tuples may contain attributes with the same identifiers as the enclosing tuple without conflict. The order of the fields in a tuple is not relevant. It is an error to define a tuple with a cycle of references, such as $\{a=b, b=a\}$.

Below we give an example of a tuple containing five fields with the attributes n , $radius$, $nested_tuple$, pi , and bar .

```
{ n = 42
  radius = 2 * pi * n      // = 2 * 3.1415926 * 42 = 263.8937784
  nested_tuple = {
    foo = "bar"
    enabled = false
  }
  pi = 3.1415926
  bar = nested_tuple.foo // = "bar"
}
```

2.2.5 Tuple composition

Tuples can be merged with the composition operator. For merging to be legal, the sets of attributes of the fields of both tuples must be disjoint. The composition operator is commutative and left-associative.

In the example below we define tuple t_3 to be the composition of tuples t_1 and t_2 .

```
t1 = {
  a = 1
  s = {
    x = 1
  }
}

t2 = {
  b = 2
  s = {
    y = 2
  }
}

t3 = t1 + t2
```

Tuple t_3 equals to:

```
t3 = {
  a = 1
  b = 2
  s = {
    x = 1
    y = 2
  }
}
```

2.2.6 Tuple inheritance

A tuple can be defined to inherit one or more tuples. When tuple s inherits from tuples t_1, t_2, \dots, t_n , we refer to s as the *inheriting* tuple, and tuple t_i as a parent tuple.

The result of the inheritance construct is that the inheriting tuple includes all the fields of the parent tuples (fields that were declared as *local* in the inherited tuple(s) are excluded). An inheriting tuple can modify the value of an inherited field by redeclaring this field and assigning it a new value. The inheriting tuple is also allowed to define new fields and exclude inherited fields. All these amendments can be applied recursively to nested tuples. Note that these amendments have no effect on the parent tuple(s); these remain always unchanged.

In the example below we define t_3 to inherit from t_1 and t_2 . In this example we modify field a that originated from t_1 , exclude field y that originated from the nested tuple s , and add a new field c .

```
t1 = {
  a = 1
  s = {
    x = 1
  }
}

t2 = {
  b = 2
  s = {
    y = 2
  }
}

t3 = (t1 + t2) {
  a = 3 // modifies a
  s {
    y = null // excludes y
  }
  c = 3 // adds c
}
```

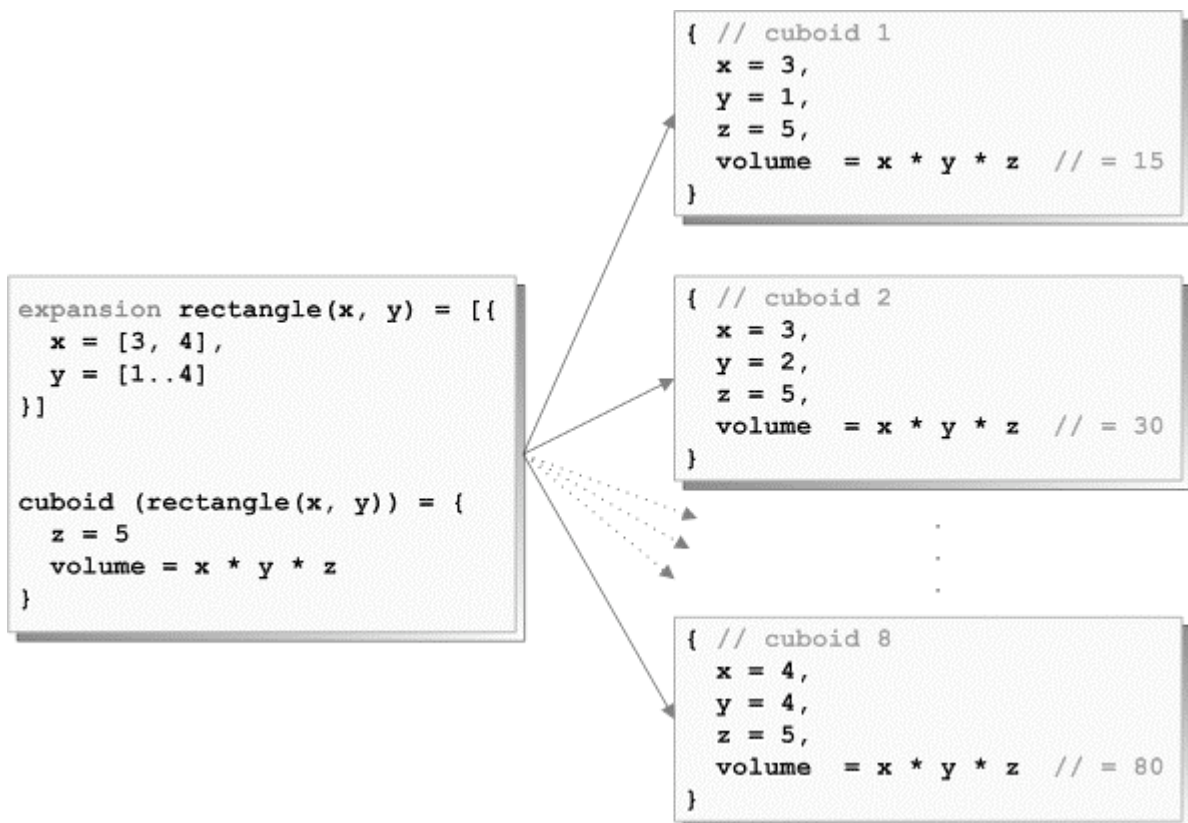
Tuple t_3 equals to:

```
t3 = {
  a = 3
  b = 2
  s = {
    x = 1
  }
  c = 3
}
```

2.2.7 Expansions and parameterized tuples

An *expansion* is a construct for generating a list of tuples. The resulting tuple list is usually used to parameterize a tuple. A *parameterized tuple* defines a collection of tuples that vary in the given parameter(s). The range of these parameters is specified with an *expansion*. Parameterized tuples are typically used to create a set of variations of some base configuration.

In the example below we use a parameterized tuple to define a set of cuboids that vary in the x and y dimension. The expansion construct is used to specify the range of the parameters x and y . The result is a collection of 8 cuboids, ranging in volume from 15 to 80.



2.2.6 References

GCL offers reference constructs that can be used to refer to the value of some other field in the configuration. GCL supports the following types of reference: *identifiers*, *up-references*, *super-references*, and *absolute-references*.

An *identifier* allows you to refer to the value of an arbitrary field in the configuration. The referred field is identified by its *selector*. A selector is a sequence of field names separated by dots, specifying the path from the root of the configuration to the field.


```

s = {
  a = 1
  b = {
    x = 2
    y = {
      u = 3
      v = 4
    }
  }
}

t = {
  k = 111
  l = s.b.y.u // = 3
  m = {
    p = s.a // = 1
  }
}

```

An *up-reference* declared inside a tuple, is used to refer to the value of a field defined in the enclosing scope of that tuple. It is permissible to chain up-references, as is illustrated in the example below.

```

a = 0

x = {
  a = 1
  b = {
    a = 2
    p = a // resolves to a = 2
    q = up.a // resolves to a = 1
    s = {
      a = 4
      b = up.up.up.a // resolves to a = 0
    }
    t = {
      b = up.s.a // resolves to a = 4
    }
  }
}

```

A *super-reference* is used to refer to a value defined in the parent tuple. Unlike up-references, super-references cannot be chained via dots.

```

t = {
  a = 1
  b = a
}

s = t {
  a = 10 // we modify the value of a
  c = super.a
}

```

With an *absolute-reference* we can refer to a field relative to the configuration file, as is illustrated in the example below.

```
a = 0

t = {
  a = 1
  b = a // resolves to a = 1
  c = @a // resolves to a = 0
}
```

2.2.7 Imports

With the import construct we can include GCL code from other GCL files. The imported file is required to be a valid GCL file. The import construct provides access to the import file via an attribute (identifier) that is specified in the import declaration.

Consider the following file:

```
// contents of file "/home/user1/gcl/project1/base.gcl"

x = 100
y = false

t = {
  s = {
    a = 0
  }
  b = 1
}
```

With the import construct we can import this file into another file, assign it an identifier, and use it throughout the configuration as if it was a locally defined tuple:

```
import "/home/user1/gcl/project1/base.gcl" as base_config

p = {
  u = base_config.x
  v = base_config.t {
    b = 5
  }
}
```

2.2.8 Templates

A tuple can be declared as a *template*. The purpose of such a template is to specify general characteristics that can be reused by specialized tuples (that inherit from the template). A tuple that is declared as a template is allowed to contain *external* fields. Such a field does not specify a value, but lets this to be filled out by an inheriting tuple. Templates are typically used to define base configurations that are reused and refined into specialized configurations.

In the example below we define the tuple *triangle* to be a template. The sides of the triangle are declared as *external* and remain unspecified in the template. Tuple *isosceles* inherits the template and assigns values to the external fields of the template.

```
template triangle = {
  a = external,
  b = external,
  c = external
}

isosceles = triangle {
  a = 1
  b = 1
  c = 1
}
```

2.2.9 Assertions

Assertions can be used to perform sanity checks in configurations. GCL offers two constructs for formulating assertions in a configuration. With the *assert* construct the user can specify a condition that emits a custom error message when violated. The *expect* construct is similar except that it emits a warning message. Assertions are especially useful in the context of templates. They can be used to specify conditions that have to be met by inheriting tuples. We give an illustration of this in the example below.

```
template triangle = {
  a = external,
  b = external,
  c = external
  assert c < a + b
}

isosceles = triangle {
  a = 1
  b = 1
  c = 1
}








bad_triangle = triangle {
  a = 1
  b = 1
  c = 3
}
```

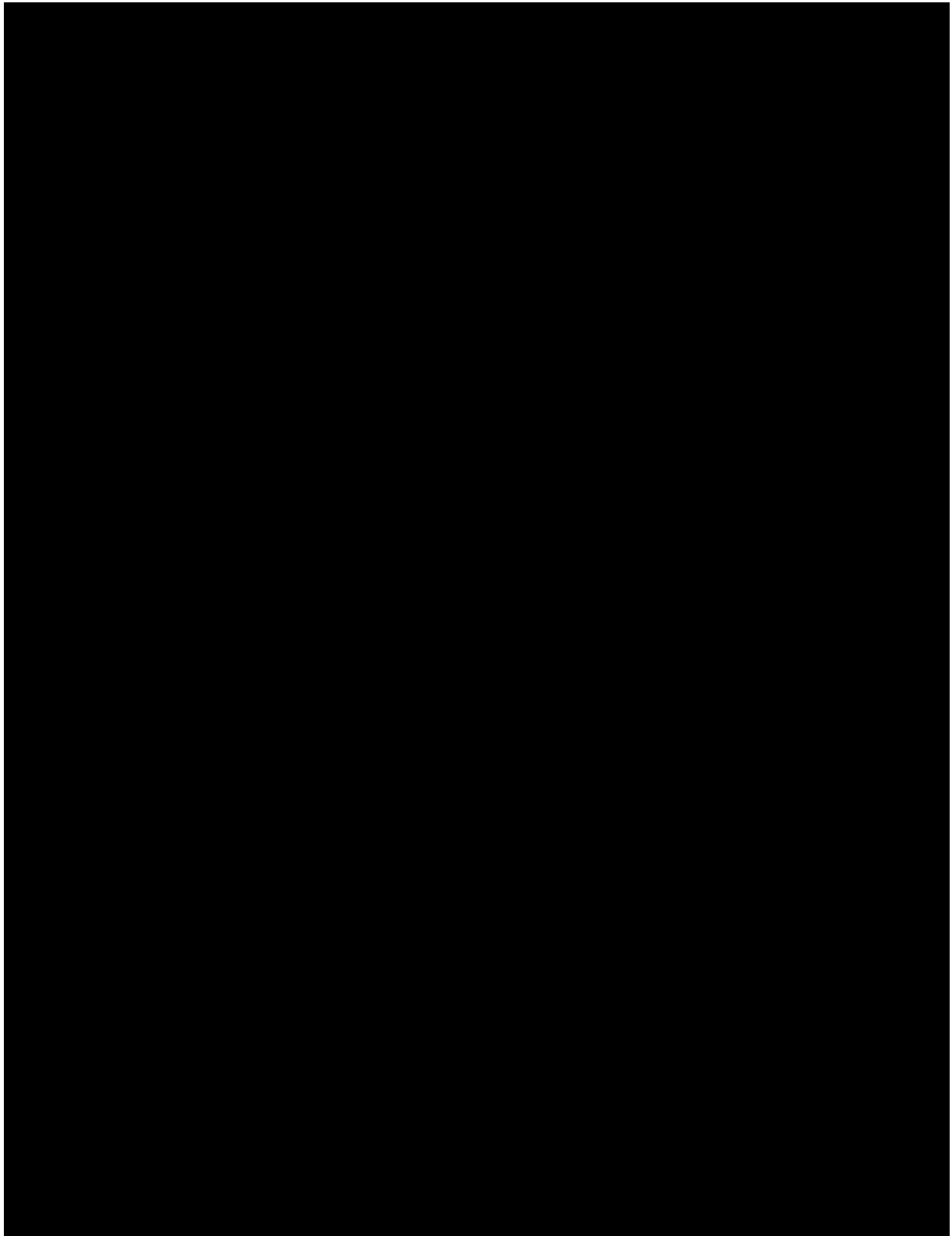
Tuples *isosceles* and *bad_triangle* equal to:

```
isosceles = {  
  a = 1  
  b = 1  
  c = 1  
  assert c < a + b // resolves to 'true'  
}  
  
bad_triangle = {  
  a = 1  
  b = 1  
  c = 3  
  assert c < a + b // resolves to 'false'  
}
```

2.2.10 Objects: Typed tuples

GCL allows you to assign a type to a tuple. A typed tuple is called an *object*. The possible types are application specific. These objects usually form the building blocks of a configuration. ■





3. Analysis and motivation

In this chapter we provide an analysis of the complexity of GCL and GCL programs. Specifically, we study which concepts of GCL contribute to the difficult understanding and maintenance of GCL configurations. The analysis serves two purposes. First, it forms the motivation for the design of our tool, and second, it serves as a guideline for the design of our tool, which we present in chapter 4.

We start with a historical outline of GCL, in which we describe how the language and the configurations have evolved in complexity over the years (section 3.1). We sum up herein some key factors and developments that have contributed to the complexity issues of GCL configurations as we know them today. In section 3.2 we describe the complexity problems of GCL configurations and capture them into a set of key issues. These key issues have formed as a guideline for the tool that has been designed as a solution to the problem. The majority of the key issues have been identified through the continuous feedback we receive from GCL users, while some have been gathered in a series of meetings between the GCL team and the core users of GCL.

3.1 Historical outline

In this section we provide a historical look on GCL highlighting important factors and developments that have contributed to the complexity of GCL (configurations).

3.1.1 Evolution of complexity

GCL was developed as an answer to Google's need for a configuration language that is secure and capable of handling its ever growing software and hardware needs. A major design goal of GCL was a concise language. The initial feature set of GCL was kept minimal, leading to comprehensible and maintainable configurations. This led to a quick acceptance and a wide spread of GCL among the engineers. However, the rapid acceptance together with the extreme growth of the company led to a much faster growth of the user base than was foreseen. Statistics show an almost exponential growth in the number of users, configurations, and lines per configuration, since the introduction of GCL in 2004 (see figure 3.1). This growth accelerated a series of developments that ultimately resulted in a drastic increase of complexity of GCL in a short period of time. The three most important developments are:

- Some language concepts that were manageable at the beginning, when the configurations were still relatively small in size, became problematic as the configurations started to scale and the users were becoming more advanced. Lambda functions together with basic arithmetic/logic/string operations were combined into long and complex expressions, for example. Another example is inheritance. As the users got more experienced they applied the inheritance construct more intensively, i.e. using deep nested (recursive) inheritance and multiple inheritance. This development increased the complexity of configurations as it has a negative effect on the readability and comprehensibility of configurations, as we will see in the next section.
- Along with the growth of the user base came the increasing habit of sharing and reusing configurations. This practice led to a drastic increase of dependencies between configurations.
- As GCL was becoming the common configuration language within the company, it became integrated with other tools and it got applied in other domains. This led to a

demand for new features in GCL, which next to the natural maturing of the language, led to an expanded feature set.

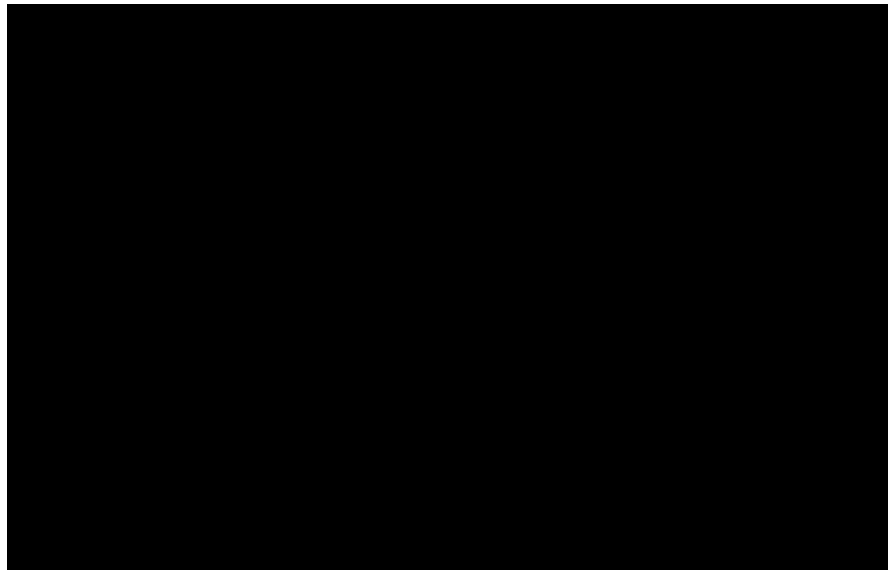


Figure 3.1: Graph depicting the number of lines of code in the GCL code base

3.1.2 Agile software development process

Google’s agile software development process plays another factor in the evolution of the complexity of GCL and GCL configurations. According to this practice a software product is developed initially as a minimal entity, and then grown iteratively into a more advanced and mature product. Consequently, the configurations corresponding to these products go through a similar evolution in size and also complexity.

3.1.3 A growing need for tooling

Parallel to all these developments was a growing need for tooling that assists users in analyzing and validating their configurations. Users want to know what they are defining and want to verify if this corresponds to what they had in mind. The language has some subtle semantic issues that can obfuscate the meaning of a configuration (these will be introduced in the next section). This need existed from the beginning, but increased as the user base was growing and the language was maturing. Moreover, as the language was maturing and becoming more wide spread, more effort was invested in developing good educational material for GCL (e.g. wikis and tutorials). From this standpoint, a tool that can help beginners to get insight into the (semantics of the) language was desired. And level the learning curve of GCL.

3.2. Key issues

The key issues that we present here serve two objectives. On the first place they aim to capture the complexity problems of GCL into a set of concrete points. Secondly, they form a motivation for an analytical tool for GCL. Consequently, some of the issues listed here relate directly to the complexity of GCL, while some describe analytical or validation needs of GCL users.

3.2.1 Complex expressions

Expressions provide a powerful and flexible way of describing the value of an attribute. But when references, lambda functions, and arithmetic/logic/string operations are combined freely this can lead to complex expressions. What makes an expression complex is that it becomes intractable for the human reader. The reason they become intractable is that a user cannot immediately determine the actual value of the attribute, which is articulated by the expression.

This is not only important for understanding what the configuration is actually defining, but it also has a validation purpose.

These complex expressions generally inhibit the comprehensibility of a configuration.

Two matters can make an expression complex (and hence problematic): references and operands. An expression is allowed to contain a reference to a value that was defined elsewhere, not even necessary in the same configuration. In order to determine the value of the expression, the user will have to resolve this reference first. This means that the user would have to locate the definition of the referenced value, which is a tedious and time consuming task. Note that the referred value could also consist of an expression, which would start another iteration of the search process. Secondly, an expression can be constituted of (self defined) lambda functions and arithmetic/logic/string operations. These operations can be easily combined into an expression that is intractable for the human reader. Note that intractability is enhanced in the cases where the user has to read and understand a configuration that was written by another user.

In figure 3.2.1 we give some examples of complex expressions.

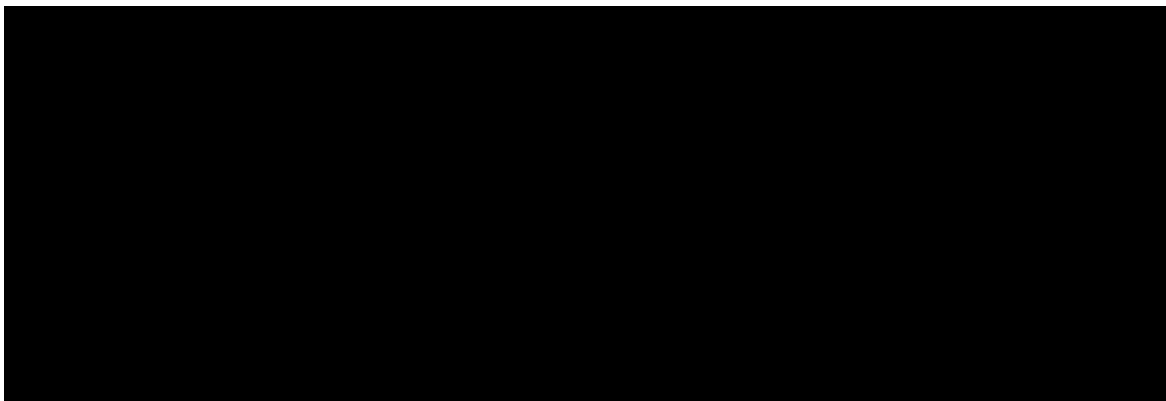


Figure 3.2.1: Examples of complex expressions

Consider the example of the variable *ram*. In order to determine the actual value of this variable, the user has to resolve the references *server.requirements.ram* and *installer.requirements.ram* in the expression first. And in case the conditional evaluates to false, the user will have to resolve another reference (*logsaver.requirements.ram*), and also look up the definition of the lambda function *optimize*, and determine what value this function returns when applied to the argument *false*. Also note that *logsaver* could have been a tuple imported from an external configuration, which would make resolving the reference even more time consuming.

Another problem with long and complex expressions is that they are more prone to human errors (e.g. mistyping). If the user would have a means to easily determine the value of the expression it will provide the user with a means of validating his expressions.

3.2.2 Unclear definition of a configuration

GCL provides an inheritance construct that makes it possible for a tuple to extend an existing tuple. This promotes the share and reuse of configurations, as it allows the user to modularize configurations. But a consequence of this construct is that an inheriting tuple includes variables that were defined elsewhere (not necessarily in the same file). This makes it in some cases hard to see what a tuple is actually defining. To illustrate this, we will introduce the two different views on a tuple: the *definition view* and the *expanded view*. The *definition view* is the way the tuple is textually defined (as it appears in an editor), while the *expanded view* is what the tuple

actually contains. The latter is a conceptual view that is obtained by resolving all the composition, inheritance and modification operations in the tuple. We denote this process as *evaluating* the tuple. Figure 3.2.2 illustrates these two views. Note that the two views are identical for a tuple that does not make use of the inheritance or composition construct.

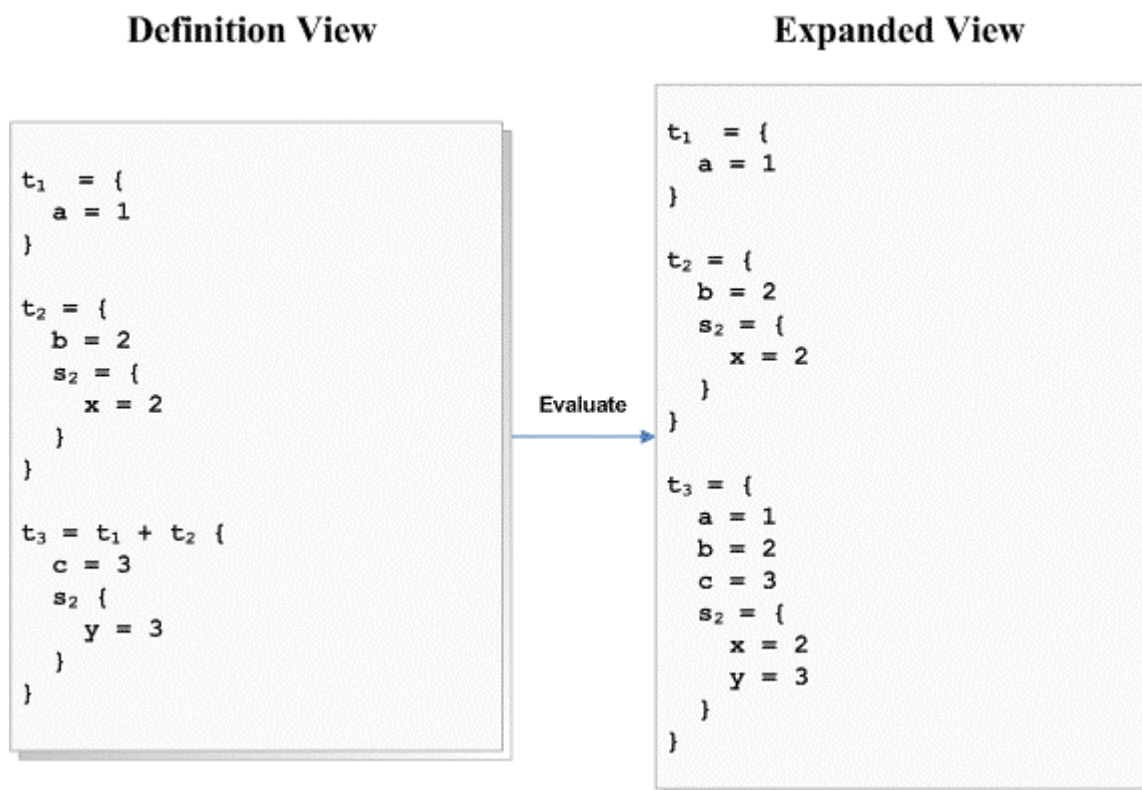


Figure 3.2.2: The definition vs. expanded view

Users have expressed that one of their main interests is to know what a configuration is actually defining. The *definition view*, however, which is what the users see in their editor, gives a distorted view on the actual configuration, as it does not show fields that are inherited. This could for example mean that the user is inheriting some parameters into his configuration without being aware of it.

In order to determine which fields the inheriting tuple actually consists of, the user has to locate the definition of the inherited tuple(s) and add (conceptually) its fields to the inheriting tuple. Although in the example above the inherited fields could just be read from the tuples defined above t_3 , one must realize that in practice a typical configuration spans thousands of lines. Moreover, GCL supports multiple inheritance, allows nested inheritance, as well as overwriting inherited values (see 3.2.5), which can lead to cases where determining the actual definition of a tuple becomes even more problematic.

3.2.3 Unknown origin of a variable

The inheritance construct is typically used together with the import construct. A common practice is to specify base configurations in separate files (templates), and then to (re)use those in an actual configuration via the import construct. With the import construct all the tuples in the template become available to the user, and using the inheritance construct the user can include one or more of these tuples into his configuration. Optionally, the user can use the modification operator to change the values of the parameters in inherited tuple. Figure 3.2.3a shows a simple

example where file 1 is imported by file 2, after which tuple t_1 is inherited and extended by t_2 . File 3, on its turn, imports file 2, after which tuple t_3 inherits and extends t_2

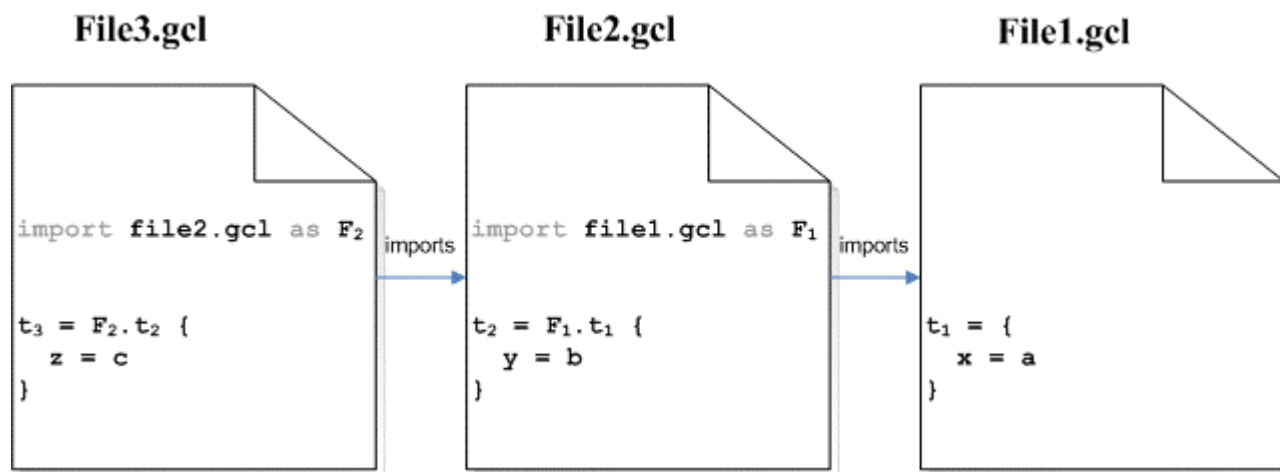


Figure 3.2.3a: An example of transitive imports

As a first observation we note that variable x is inherited indirectly by tuple t_3 . This forms another motivation for the *expanded view*, because from the definition of t_3 it does not follow directly that it also inherits a variable from t_1 . This makes tracing back the actual definition of t_3 an even more time consuming task. Moreover, in practice this could mean that writer of the configuration in *file 3* could have pulled in a parameter x without being aware of that.

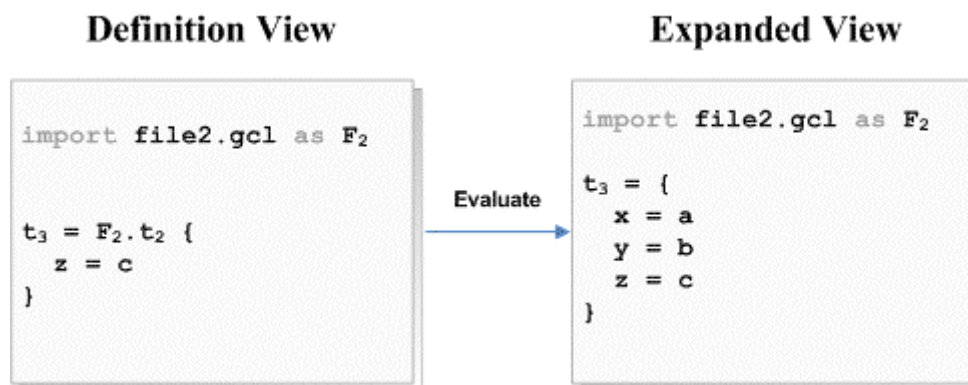


Figure 3.2.3b: The two views on imported values

There is another issue that comes along with the import construct. When we present the *expanded view* in figure 3.2.3b, the information of where a field was originally defined is missing. This information is of importance when a user wants to alter the value of a parameter, for example. In the current case the user has to manually locate the file containing the variable to be altered, and then search within this file for the line where this parameter is defined.

There is also an analytical purpose. For the maintainer of the configuration it is interesting to know which parts of the configuration come from where. If he wants to modify the original value of x , he needs to know in which file it was defined. In the current situation the user has to search through the files manually.

3.2.4 Hidden Dependencies

The import construct can be applied transitively, as we have seen in the example of figure 3.2.3a. A file x can import values from a file y , which file y has imported from a file z . As a result, a file can include fields that originate from a file that is not listed in its import section. But a file that is referencing another file has a dependency on that file. In the example of figure 3.2.3b, file 3 only imports file 2, but it also has a dependency on file 1, as it has inherited field x from it (via file 2). When the value of x is altered in file 1 it will affect the configuration in file 3. This process of transitive inclusion can be repeated several times in different directions. Consequently, it is conceivable that a file that is only importing one file directly is actually referencing a dozen other files indirectly.

This lack of transparency can make the maintenance of configurations problematic. A change to some configuration (e.g. a template) could go unnoticed to someone that is referencing that configuration indirectly. In our example, a change to field x in file 1 affects the configuration in file 3. Therefore, it is vital for the maintainer of file 3 to know on which files his configuration depends. It is important to make these dependencies transparent to user. Secondly, a total view on the dependencies of a configuration could also bring unnecessary or unwanted dependencies to the attention of the user. And this could lead on the long term to an overall reduction of dependencies between configurations. Finally, insight into dependencies can promote the modularity of configurations, and therefore make them easier to maintain and reuse. Consider for example a file x that is dependent on z indirectly via file y . If the user would realize that z is actually not needed by y , it makes sense to factor out the part of y that includes z .

3.2.5 Unintended modifications

GCL offers the possibility to modify the value of an inherited variable. Formally, when tuple s inherits a tuple t , it is legal to modify the value v_i of any field f_i defined in tuple t . To modify the value the user needs to redeclare the variable in the inheriting tuple, assigning it a new value (fields marked with the *final* keyword are an exception since they are prohibited to be redefined). The result is that the initial value of the variable is overwritten with the new value. Figure 3.2.5 depicts an example of such a modification. In this example tuple t_2 inherits from t_1 , and modifies variable a . Its original value 1 becomes overwritten by the new value 2.

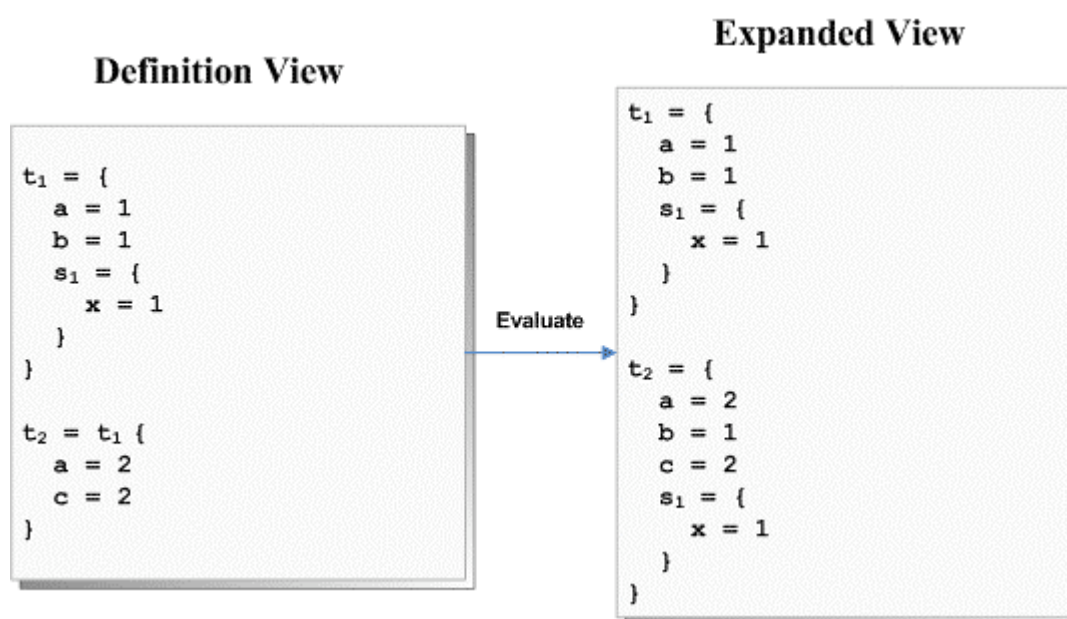


Figure 3.2.5a: An example of a modification

The modification construct is useful in the context of sharing and reusing configurations. It allows a user to adapt some parameters defined in a base configuration (e.g. default values in a template). However, the use of modifications can become problematic when a user modifies a variable unintentionally. One must realize that a typical configuration spans thousands of lines, and since they are usually applied in a specific domain, the names of parameters are chosen from a common vocabulary. It is therefore not unlikely that a parameter will become redefined accidentally in an extended configuration. And since it is a legal language construction, such a modification will not trigger a warning or error message. In order to prevent these unintended modifications a user would be expected to search through all the inherited configurations manually. And as we have seen before, these configurations could on their turn include other configurations, making it a time consuming and an error prone effort (i.e. a user could overlook some fields).

The modification construct is tricky for another reason. A modification of a variable x could affect the value of other inherited variables. In example 3.2.5b, the modification of variable a affects the value of variable b in tuple t_2 .

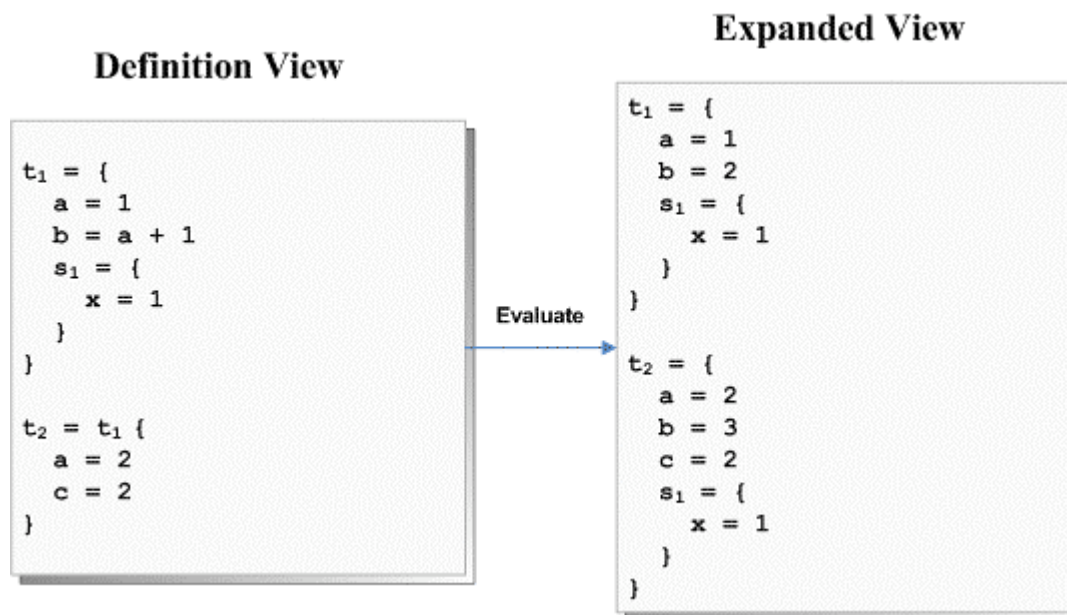


Figure 3.2.5b: An example of an indirect modification

3.2.7 Cascaded inheritance

There is another issue that comes along when the inheritance construct is applied intensively. Users have reported to find it in some cases hard to keep track of the value of an inherited parameter. They also have stated not always to understand how such a parameter got the value it has. When the inheritance construct is applied iteratively, a parameter is passed on through a chain of tuples before ending in the final tuple. At each stage of this process the parameter can be assigned a new value. In figure 3.2.6 we illustrate this with a simple example.

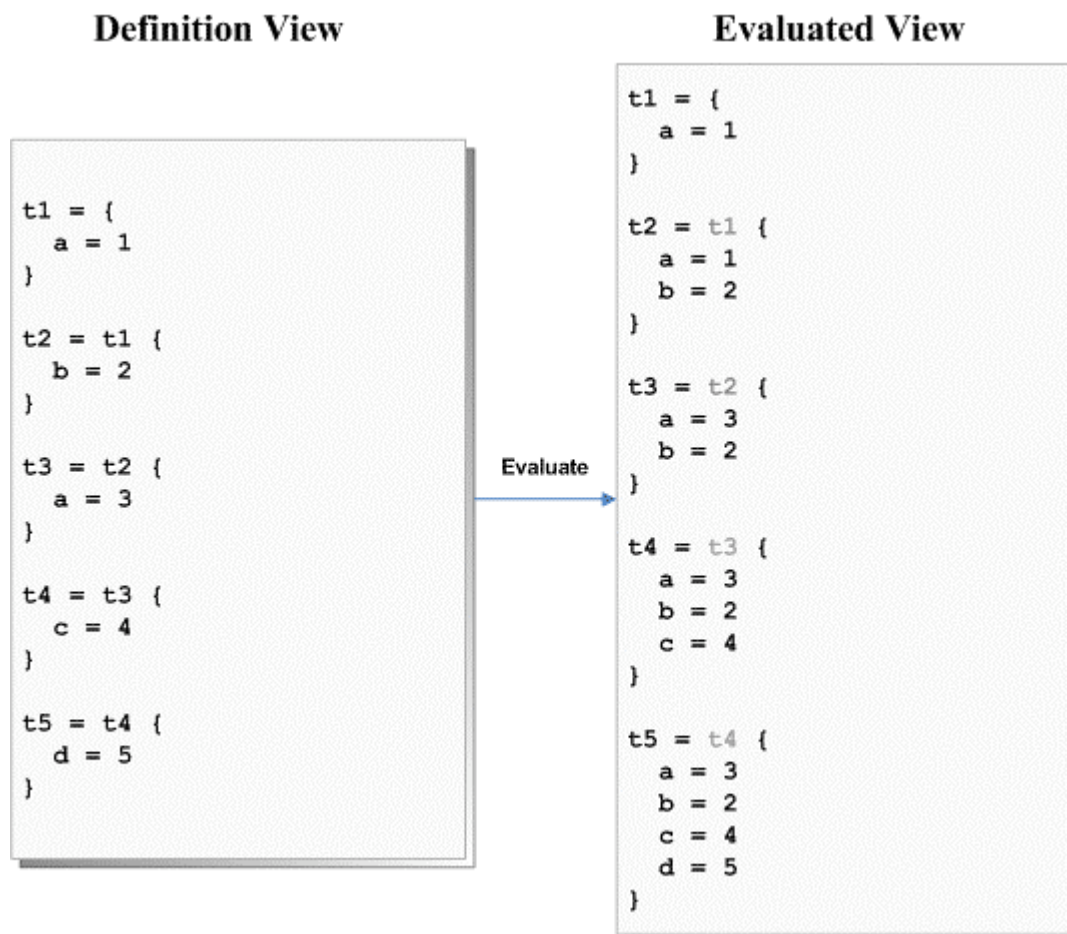


Figure 3.2.6: An example of cascaded inheritance

Consider the parameter a in tuple t_5 , although it was originally defined in t_1 and inherited from t_4 , it received its final value of 3 in t_3 . It is meaningful for a user to have insight in this process. For example, when a parameter has a different value than expected, the user will be interested where that unexpected value was assigned. Moreover, the information of the place and the context of the modification can help the user understand the why the variable was assigned that value.

Note that in practice this issue occurs in the context of tuples that are cluttered with hundreds of other fields, making this more intractable. Moreover, when inheritance is applied hierarchically (with nested tuples) this leads to even more complex inheritance issues.

Further, note that the issue described here is different from the issue of the origin of a parameter described in 3.2.3. In our example, the origin of parameter a in tuple t_5 is tuple t_1 (the place where it was defined), while its final value was assigned in tuple t_3 .

3.2.7 Lack of overview in a configuration

In a production environment GCL configurations span thousands of lines. In some cases the user only wants to have a general understanding of a configuration, and is only interested in the structure and the key elements of the configuration. Tuples and especially objects (typed tuples) provide a good guideline for the essence of a configuration. But a typical configuration is cluttered with thousands of other fields, obscuring this general picture of the configuration.

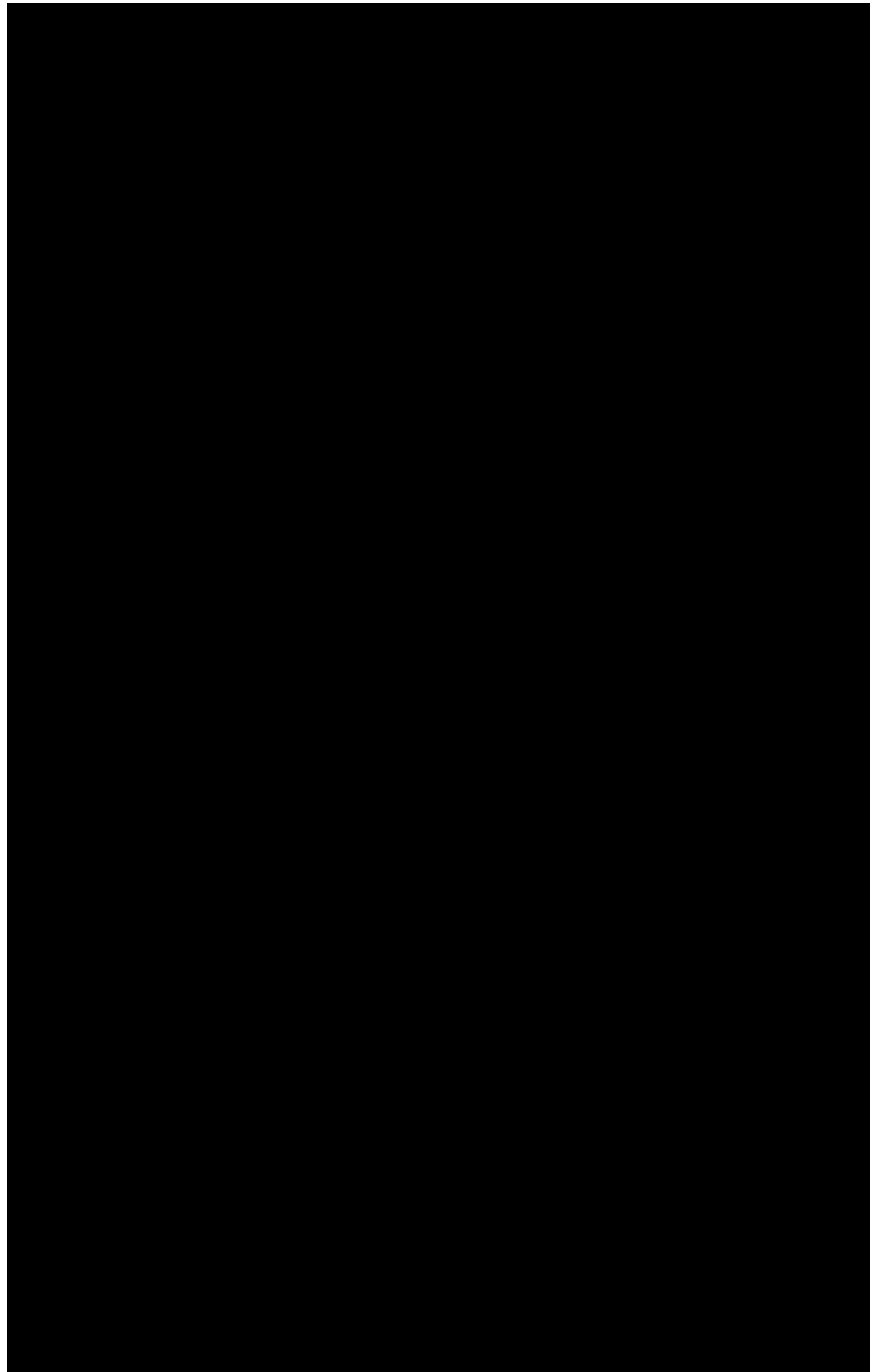


Figure 3.2.7: [Redacted]

3.2.8 Erroneously defined parameterized tuples

In the previous chapter we introduced the notion of a parameterized tuple. This construct allows us to generate a collection of tuples that vary in the specified parameter(s). The users have expressed to be interested in seeing the collection of tuples that results from such a parameterized tuple. In a text editor there is no way to see these tuples. There is also an important validation issue. The expansion construct provides a powerful way of generating

thousands instances of a tuple with a single declaration. A mistake in the boundaries of the range of the parameters can lead to an unintended amount of replication of a tuple. One could create several thousands instances of a configuration accidentally. This can lead to an intensive resource consuming configuration. The user needs therefore to get insight in the result of the expansion operation, as a means of validation, before deploying the configuration.

3.2.9 Ambiguous references

GCL offers several constructs for creating references within a configuration. An example of such a reference is the *super* reference, which can be used to let a variable refer to some field in the parent tuple. The result of this operation is that the referring variable will take the value of the referred field in the parent tuple.

For the understanding of a configuration a reader needs to know how such a reference resolves. The references have each their own way of resolving and are also subject to scoping rules. This can lead to ambiguous seeming situations, especially for the novice user. In the example of figure 3.2.9, the reference of variable *c* resolves to the variable *d* defined outside the tuple (at the top of the configuration), but since tuple *t₃* contains a variable *d* itself (inherited from *t₂*), a user could become confused and think that it resolves to this *d*.

```
d = 0

t1 = {
  a = d // resolves to 0
}

t2 = {
  d = 1
}

t3 = t2 {
  a = 2
  b = super.d // resolves to 1
  c = d // resolves to 0
}
```

Figure 3.2.9: An example of ambiguous references

But also when the user does know these rules, resolving references can be problematic. Resolving references in large files is a time consuming task. The referred value could be defined several hundreds lines further and the user has to search for it manually while adhering to the scoping rules.

4. Tool design

In this chapter we present our tool - GCL Viewer - and describe its design. We start with a general description of the tool (section 4.1), followed by an outline of the design process (section 4.2) and the methods and techniques that we have used (section 4.3). In section 4.4 we present the main features of the tool. These features have been designed to deal with the key issues identified in the previous chapter. In section 4.5 we give an overview of additional features of GCL Viewer.

4.1 GCL Viewer

GCL Viewer is a software visualization tool [Price93]. The tool aims to improve the readability and comprehensibility of GCL configurations. The tool can be used to read and browse through GCL configurations, and it provides a means of analyzing and validating GCL configurations. The tool has been designed to work with GCL and languages derived from GCL (e.g. ██████). However, the technical design of the tool (chapter 5) facilitates the reuse of the tool for other languages. In the remainder of this section we describe the key characteristics of GCL Viewer.

Explicit view

The main characteristic of the tool is that it provides an explicit view on a GCL configuration, showing what the configuration is actually defining. To clarify this notion, we recall from chapter 2 that in essence, a GCL configuration can be perceived as a structured collection of attribute-value pairs. Language constructs such as inheritance, imports, expressions and parameterized tuples can obfuscate this perception, especially when applied intensively. GCL Viewer resolves and unfolds these constructs, and shows the resulting structured collection of attribute-value pairs. A key feature of the tool that contributes to the explicit view is the *expanded view* (section 4.4.2), which flattens tuple definitions and shows the tuples including their inherited and imported content. *Evaluate expression* (section 4.4.1) provides a way to resolve expressions and see the computed value of a parameter. Another important feature is *browse parameterized tuples* (section 4.4.9), which shows the collection of tuples that result from a parameterized tuple.

Analytical and validation functionality

Another characteristic of GCL Viewer is that it provides analytical and validation functionality. The analytical functionality is designed to assist in getting a deeper understanding of GCL code. Examples are the *field inspector* (section 4.5.4), which provides an overview of properties of a variable, and *dependency listing* (section 4.4.4.1), which give insight in the relationships between configurations. Examples of validation features are *evaluate expression*, which provides a way to validate expressions in a configuration (*report errors and warnings* (section 4.5.2) informs about mistakes in expressions), and *browse parameterized tuples*, which can be used to verify definitions of parameterized tuples.

Enhancing readability

GCL Viewer incorporates a set of visual techniques that aim to improve the readability of GCL configurations. *Text markup* (e.g. syntax coloring, indentation), *foldable tuples*, and *truncated expressions* are the most important ones (section 4.5.1).

Code browsing

GCL Viewer can also be used as a code browser. References between configuration files are displayed as hyperlinks (section 4.4.3.2 and 4.4.4.1) allowing easy navigation through the code base. Also GCL references (e.g. up-reference, super-reference), which are used to refer to values within a file, are presented as hyperlinks (see section 4.4.9)

Interactive

GCL viewer is an interactive tool. It provides interaction mechanisms through which the view on the configuration can be manipulated (e.g. evaluate expressions, fold/unfold code sections) and which can be used to obtain additional information about a configuration (e.g. field inspector). GCL Viewer provides these interactions via both a pointing device and a keyboard (see section 4.5.5).

Web integration

An important characteristic of GCL Viewer is that it facilitates the integration of the tool into web documents. This offers the possibility to enrich online software documentation (e.g. tutorials, wikis) and web applications (e.g. monitoring systems, discussion boards) with this tool, taking advantage of its enhanced readability and understanding capabilities, and its analytical and validation features.

We believe that visualization tools such as GCL Viewer can assist in teaching a programming language. These tools can provide a different insight on the teaching material when integrated into tutorials and wikis for programming languages. Tutorials and wikis typically contain source code examples, which could be visualized by our tool to provide a better understanding of it. Our tool can also be used to provide interactive teaching material. A possible set-up is sketched in figure 4.1a. In this concept, the user can edit the code examples and use our integrated viewer to see the effect. The user can also experiment with own GCL code (fragments) and see the result instantaneously.

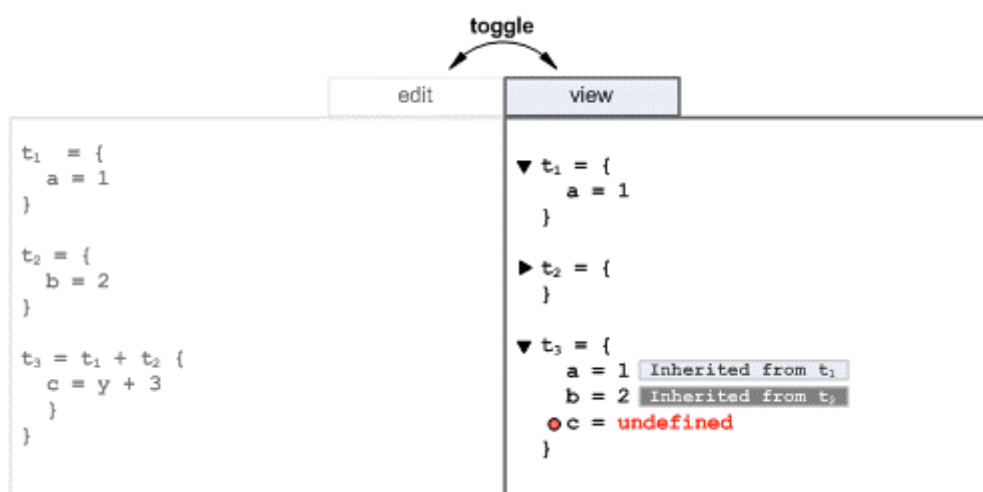


Figure 4.1a: An interactive application of GCL Viewer

UI-modes of GCL Viewer

GCL Viewer is available in three UI-modes, each offering a different appearance of the tool: the *standard* mode, the *inlined* mode, and the *mini* mode. The standard mode presents a complete user interface, including a menu and logos (as is shown in the screenshot in Figure 4.1b). The inlined mode displays only the visualized source code, which is useful for integration into documents (Figure 4.1a is an example of the inlined mode). The mini mode is a rescaled version of the standard mode; it uses a smaller font and smaller icons to present a compact form of the tool. This mode is useful for integration of the tool into (online) dashboards.

Screenshot

In figure 4.1b we present a screenshot of GCL Viewer. The elements of the user interface will be explained in detail in sections 4.4 and 4.5. For more screenshots we refer to appendix D.



Figure 4.1b: A screenshot of GCL Viewer

4.2 Design process

We have designed GCL Viewer according to an iterative design process. Through rapid prototypes we were able to communicate our ideas to the users and receive early feedback on the design. Throughout the design process we have also discussed our design proposals with the User Experience team at Google.

4.3 Methods and techniques

4.3.1 A visual tool

We choose to design GCL Viewer as a visual tool. This allows us take advantage of the remarkable human perceptual ability for visual information [Shneiderman98]. A visual tool allows us to use visual techniques such as color coding and highlighting, to carry information, portray relationships, draw attention to a particular item in a mass of thousands of items, among others. In the following section we will give a detailed description of the visual techniques that we have applied in the tool.

4.3.2 Visual techniques

We use Shneiderman's Visual Information Seeking Mantra [Shneiderman98] to describe the visual techniques that we have applied in the design of GCL Viewer. Our main design principle is to comply with the design rule advocated by Shneiderman: Overview first, zoom and filter, then details on demand. We will explain and motivate the cases where we chose to deviate from this principle.

The initial view in GCL Viewer is an expanded view of the configuration where the entire collection of attribute-value pairs is presented. We deviate here from Shneiderman's guideline to give an **overview** of the collection first. In a first prototype we showed only the top level tuples of a configuration. By unfolding these tuples the user could reveal the rest of the configuration step by step. From the early user feedback that we received on this, we have learned that users have a preference for a complete view on the configuration (as the initial view). This corresponds to the earlier identified interest of users, stating that users primarily want to know what a configuration is actually defining. The tool does offer overview options on demand. The *skeleton view* (section 4.5.7) shows a tree structure of all tuples in the configuration. A second overview option is the *list key tuples* feature (section 4.5.8), which provides a listed overview of the main tuples in the configuration.

The *select key tuple* feature allows us to **zoom** on certain sections of the configuration. GCL Viewer offers two **filtering** mechanisms. The *skeleton view* (section 4.2.9) described before, removes all the fields that are not tuples from the visualization. The *hide inherited content* filters out all the imported and inherited content, yielding the original definition of the configuration. With *fold tuples* we can filter out irrelevant code sections.

The initial view shows the collection of attribute-value pairs only. Additional information is provided according to the **details-on-demand** principle. This has two advantages. It allows us to keep the user interface simple, which is important for new users. Second, it allows us to maintain a clean user interface, which is important for maintaining a good readability of the configuration. We use pop up layers and tooltips to provide additional information. We use pop up layers when we expect it is useful for the user to have the information on the screen for a longer time (e.g. *field inspection* (section 4.5.11)). Tooltips are used for information that needs to be accessed quickly and for a short time (e.g. *origin info* (section 4.5.3)). We use tags and icons as visual cues to indicate the presence of additional information, e.g. a red button in front of a field to indicate the presence of an error message.

GCL Viewer uses color coding and textual markup to **correlate** elements of the configuration. *Syntax coloring* (section 4.5.1) is used to identify and associate the tokens in a configuration (e.g. tuples, expressions, references). With the *dependency coloring* feature (section 4.4.4.2) we can associate fields that originate from the same (imported) file by assigning them a highlight color. The *inheritance tags* (section 4.4.5) feature uses colored tags to correlate fields that originate from the same parent tuple.

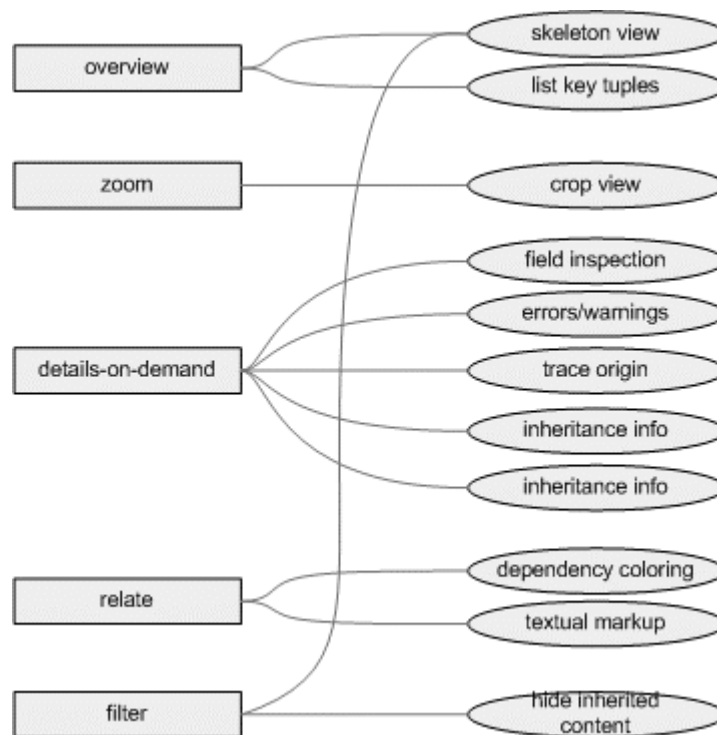


Figure 4.3: Mapping of GCL viewer features to visual techniques

4.3.3 Hypertext browsing

GCL Viewer uses the hypertext technique to facilitate browsing through GCL code. A GCL configuration can contain references within itself (section 2.2.6) or references to other configurations (imports). These references are represented as hyperlinks, which can be used to navigate to the referred location. The advantage of the hypertext technique is that it provides not only an effective but also a familiar way of navigation, due to the wide spread habit of web surfing.

4.3.4 Hybrid interaction control

The advantage of pointing is that it allows rapid selection [Shneiderman98] and it does not require remembering commands. The keyboard control allows advanced user to work more efficiently. GCL Viewer therefore offers interaction via a pointing device and keyboard. The tool can be fully controlled both with a pointing device and keyboard, meaning that the code can be browsed and that the features are accessible in both ways.

4.3.5 Color techniques

A main reason to use colors in our visualization is because color has the power to communicate a lot of information efficiently [Brown]. By using colors we can reduce the amount of textual information that is displayed, which improves the overview and readability of our visualization. In section 4.3.2 we described how we use colors to correlate elements in the visualization. We give an overview of other uses of colors in GCL Viewer:

- We use colors to communicate the presence of error and warning messages in a configuration (section 4.5.2), and to state the result of assertions (4.5.3).
- Another use of color is to highlight an area of interest; the feature *Navigate References* (4.4.9) uses a highlight color to indicate the field that is being referenced.

As for the choice of colors, we use standard colors where possible, e.g. for errors and warnings we use the colors red and orange, respectively. Similarly, for the result of an assertion we use the

color green to indicate that the assertion evaluates to *true*, and the color red for *false*. The feature *dependency coloring* (section 4.4.4.2) allows the user to choose the color from a color picker menu. As for syntax coloring of the source code: we choose the color scheme that is used in the VIM editor, which is the most popular editor for GCL programs within Google. In the future we want to extend GCL Viewer to allow the users to choose their own syntax coloring scheme.

4.3.6 Font techniques

GCL Viewer displays the source code using a fixed width font. In a fixed width font all characters have the same width. Fixed width fonts are preferred by most programmers for editing and viewing source code, as they allow text to align more readily, and make inspecting the code for errors easier (typographical mistakes are easier to spot in text that is formatted with a fixed width font). Fixed width fonts are also the default in most source code editors, terminals, and IDEs.

We use font and textual opacity variations to distinguish between locally defined and imported fields (section 4.5.1), and between the raw and evaluated form of an expression (section 4.4.1). Like coloring, the font technique allows us to convey information to the user without providing additional textual information, which helps us to maintain the overview in the visualization.

4.4 Main features

In this section we will describe the main features of the Viewer. Each of these features addresses one or more issues described in the previous chapter. We will illustrate the features with examples where necessary. Note that the numbering of the sections corresponds to the numbering of the key issues from chapter 3.

4.4.1 Evaluate expression

The viewer offers an easy way to resolve an expression. By clicking on the expression, a user can toggle between the original (denoted as *raw*) and the evaluated form of the expression. In the evaluated form, references, arithmetic/logic/string operations, and function calls are resolved (as far as possible). This generally leads to much shorter values, promoting the readability and comprehensibility of expressions. Consider the example from the previous chapter, depicted in figure 4.4.1a.

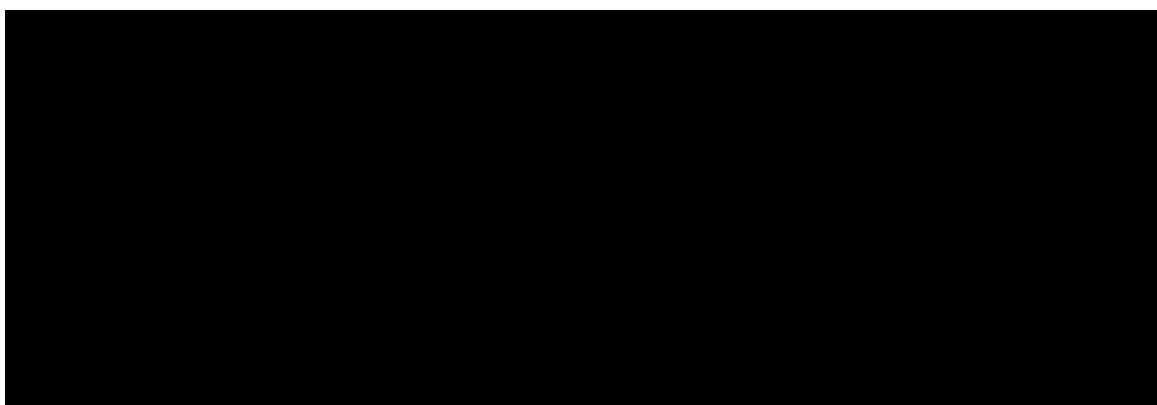


Figure 4.4.1a: Raw (unevaluated) expressions

In figure 4.4.1b we see the same example where all the expressions are evaluated. We can see how evaluating expressions leads to a more compact representation of the configuration, promoting the readability of configurations in general. An evaluated expression is displayed with a different font color than a raw expression, allowing us to distinguish between the two forms of an expression.

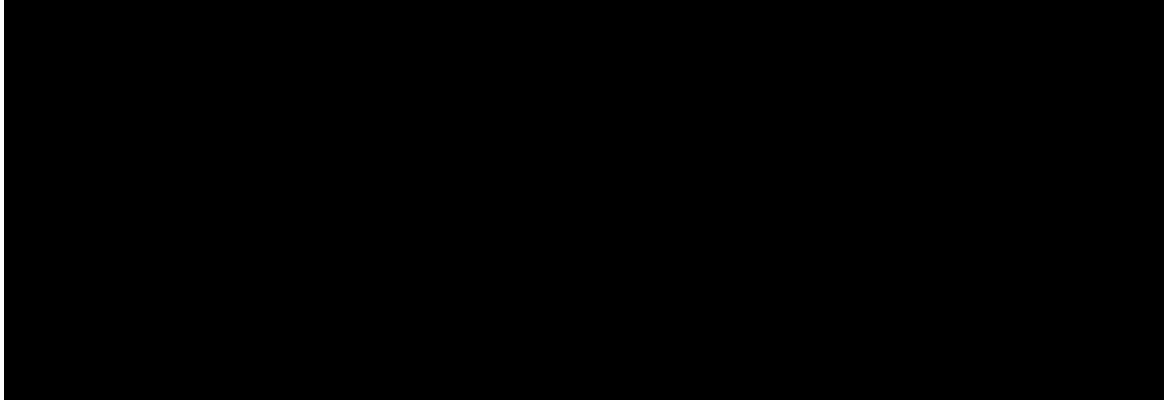


Figure 4.4.1b: The result of evaluating the expressions

Another benefit of this feature is that it provides the user a way to validate expressions. An expression that is syntactically incorrect evaluates to the value *undefined*. In such a case the viewer will also present the associated error and/or warning messages, which informs the user about the error in the expression (section 4.5.2)

4.4.2 The Expanded View

In the previous chapter we introduced the concept of the *expanded view*. This view corresponds to what a tuple actually contains, i.e. including inherited content. The viewer provides an expanded view on the entire configuration. This means that all tuples in the configuration are displayed including their inherited content. This view on the configuration provides a better picture on what the configuration is actually defining, which was one of the major concerns of the users. An illustration of the expanded view is given in figure 4.4.2.

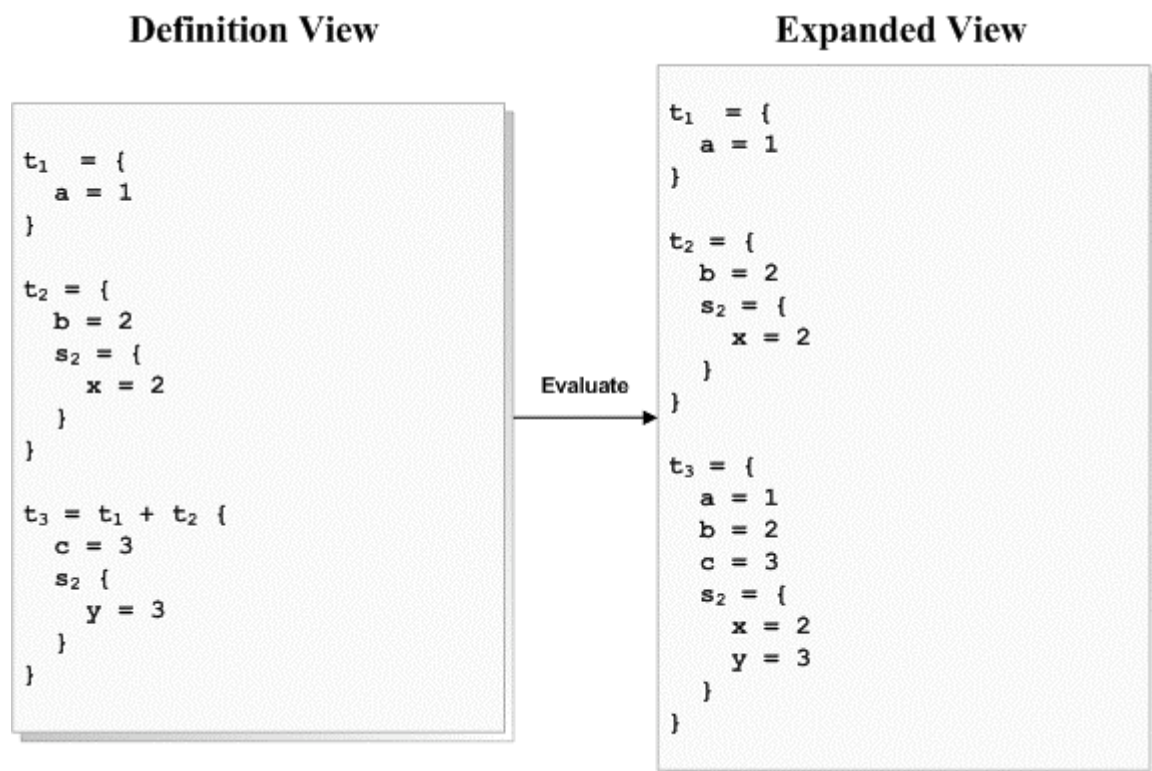


Figure 4.4.2: The expanded view

The expanded view is shown by default, but the viewer also offers the option to hide inherited content. This results in the *definition view*, which corresponds to the way the configuration looks like in a text editor. The viewer also offers a convenient way to distinguish between locally defined and inherited fields: A tag is placed behind an inherited field stating the tuple it was inherited from (see section 4.4.6.1 Inheritance tags).

4.4.3.1 The origin of a field

The viewer provides information about the location where a field was originally defined (the location is defined as a filename and line number pair). This feature is useful in the evaluated view, where imported and locally defined fields are all merged together. The origin information is presented as a tooltip to the field (see figure 4.4.3 for an example). This approach maintains the overview in the configuration, while keeping the information easily accessible.



Figure 4.4.3: The origin of a field

4.4.3.2 Locating the origin of a field

The viewer displays all inherited fields as hyperlinks (see figure 4.4.3). This allows easy navigation to the origin of the field. If the origin is located in a different file, this file is opened in a separate window. The viewer highlights the origin and centers it on the screen, which makes it easier for the user to see. Note that this also provides a way to distinguish between locally defined and inherited fields inside a tuple.

4.4.4.1 Dependency Listing

The viewer provides an overview of the dependencies of a configuration. The viewer presents a dependency list at the top of the configuration (where the imports are normally defined). Any file that is referenced by the configuration (directly or indirectly) is listed in the dependency list. See figure 4.4.4 for an illustration.



Figure 4.4.4: Dependency list

The viewer uses two different font styles to distinguish between directly and indirectly referenced files. The files in the dependency list are presented as hyperlinks, allowing the user to conveniently locate the referenced configuration. A checkbox in front of each dependency allows the user to toggle the visibility of the fields that were imported from that dependency.

4.4.4.2 Dependency Coloring

Dependency coloring is another feature that aims to give the user insight into the dependencies of a configuration. This feature provides the user an easy way to highlight all the fields that originate from a certain dependency with a user picked color. This allows the user to divide the configuration into different color regions, where each region represents a dependency. This gives a user a good overview of which parts of the configuration are dependent on which file. Figure 4.4.4.2 shows a simple illustration of this concept. In the dependency list, each line is prefixed with a color picker. Upon selection of a color, all the fields originating from that dependency are highlighted with that color. To keep the colors recognizable, the color picker is then assigned the same color.

**Figure 4.4.4.2: Dependency coloring**

4.4.5 Modification tags

The viewer uses tags to make modifications transparent. The viewer inserts a tiny tag behind a field that represents a modification. The tag contains the name of the field that is being modified. To prevent the screen from getting clogged with tags, all modification tags are collapsed initially. A click on the tag expands it and reveals the information contained. In figure 4.4.5 we present an illustration of the modification tag.

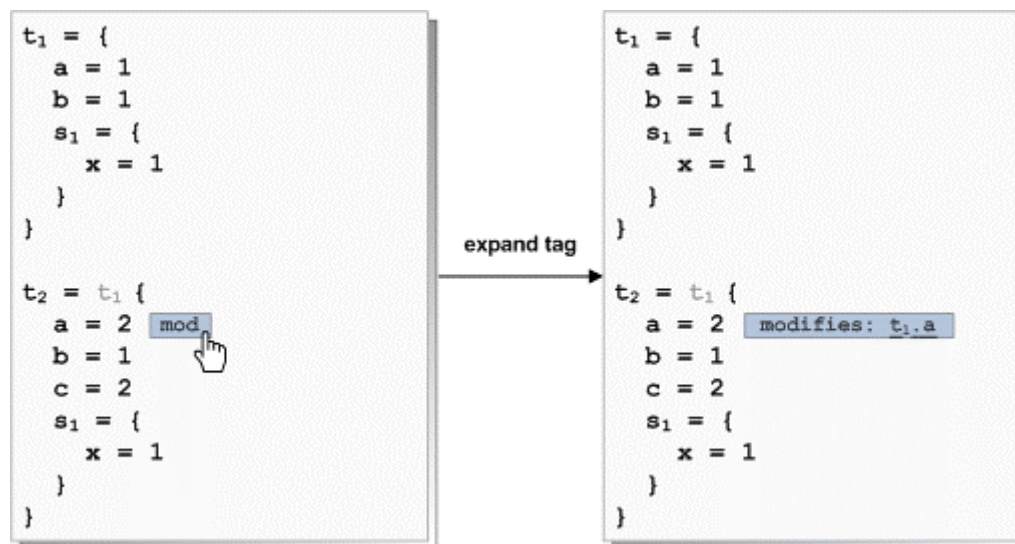


Figure 4.4.5: Modification tags

The name of the modified field in the tag is presented as a hyperlink. This allows the user to jump to the field that is being modified and inspect its old value.

4.4.6 Inheritance info

We know from chapter 2 that a tuple can inherit fields from other tuples. In section 3.2.6 we have explained how inheritance can obfuscate the meaning of a configuration and make them harder to understand. GCL Viewer provides two features that aim to bring insight into inheritance issues, *inheritance tags* and the *inheritance stack*.

4.4.6.1 Inheritance tags

As a result of applying the inheritance construct, a tuple can contain a mixture of fields that were locally defined and fields that were inherited from other tuples. The *expanded view* (section 4.4.2) shows this result by displaying both types of fields. A question that can arise from this view is from which tuple a certain field originated. GCL Viewer provides a feature that allows you to identify from which tuple a field originated from. The feature places colored tags behind each field, stating from which tuple it was inherited. In order to reduce the number of tags on the screen, we choose not to tag fields that were locally defined in the tuple. Hence, an absence of a tag means that the field was defined locally. In figure 4.2.6.1 we see the same example from section 4.4.2 enriched with inheritance tags.

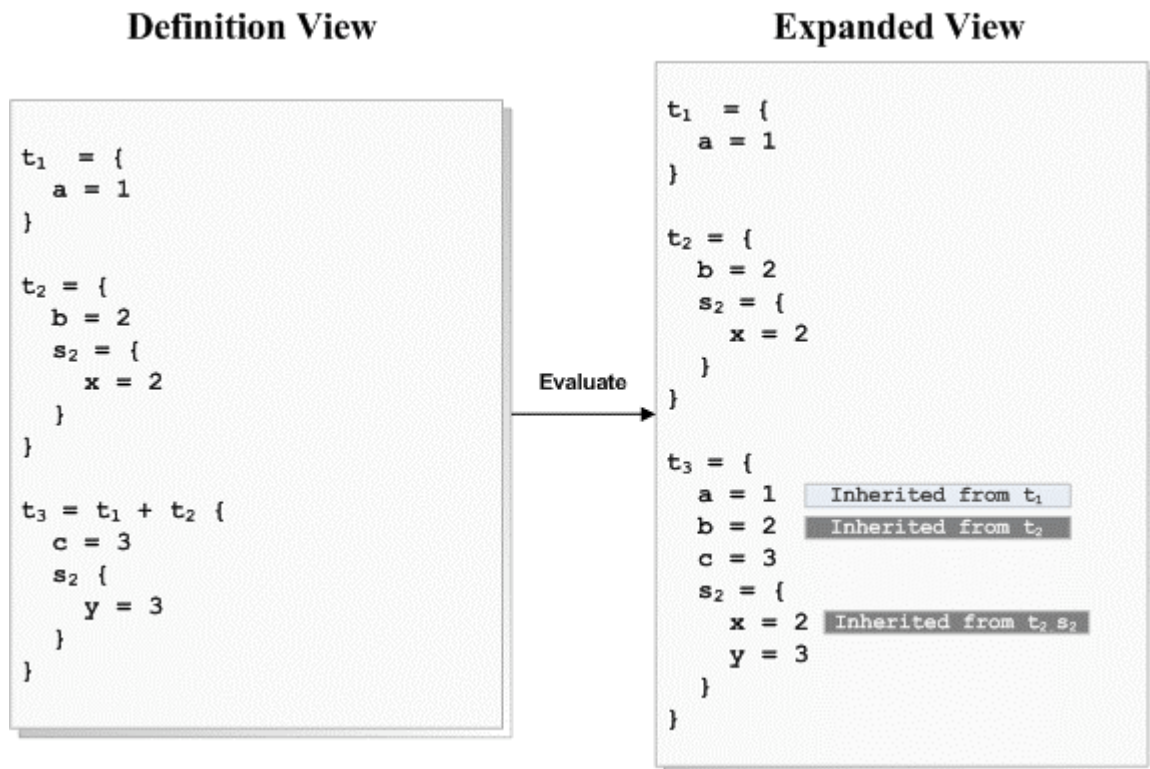


Figure 4.2.6.1: Inheritance tags

4.4.6.2 Inheritance stack

The viewer also offers a more advanced feature to provide insight into inheritance issues. The *inheritance stack* shows the inheritance history of a field. It starts with the tuple where the field was originally defined, and continues listing the tuples that have inherited the field, until the last tuple the field was inherited from. At each inheritance stage we show the value the field. This provides insight in how the value of the field evolved and where it obtained its final value. In figure 4.2.6.2 we present an illustration of this feature.

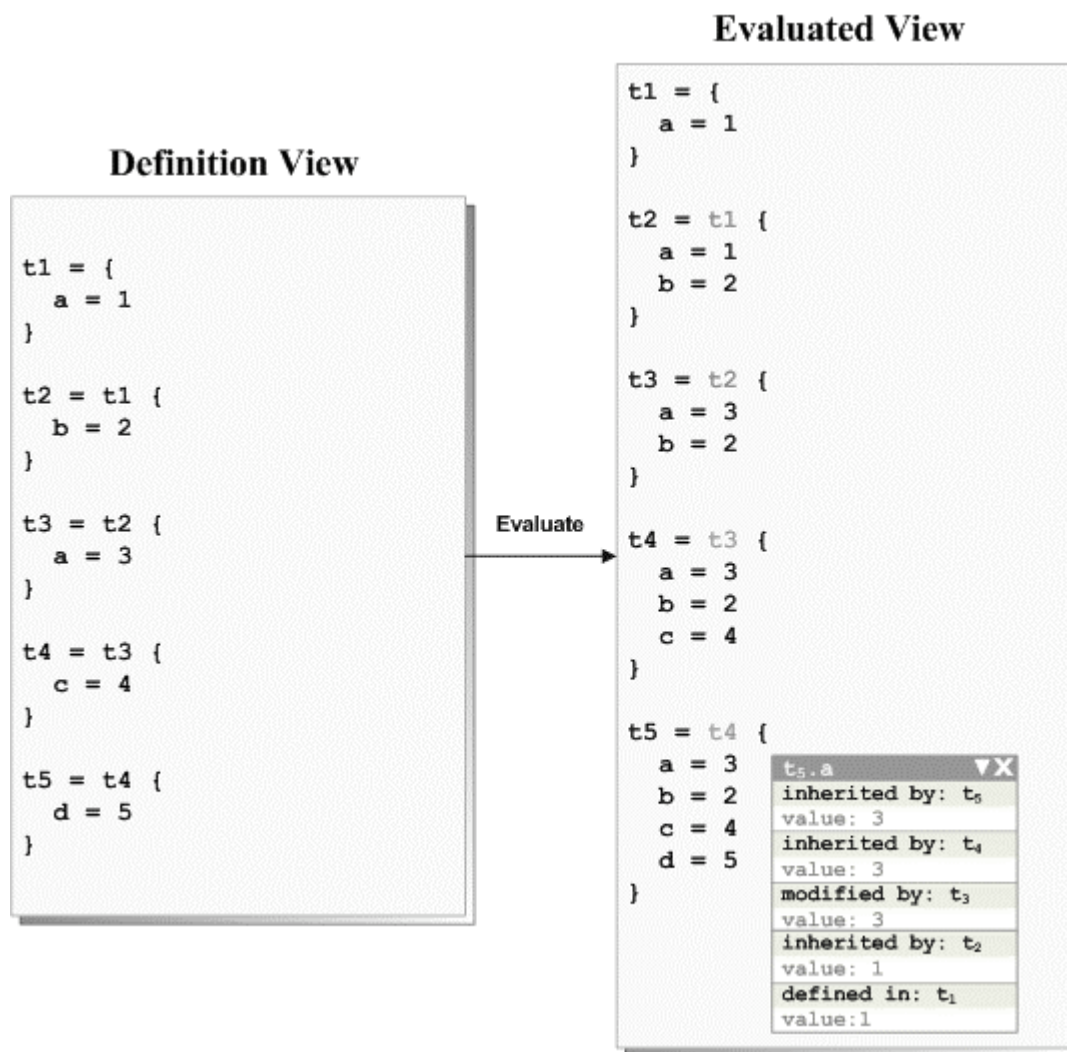


Figure 4.4.6.2: The inheritance stack

4.4.7.1 Skeleton View

For a general understanding of a configuration, definitions on the macro level are more relevant than the details of some parameter settings. Based on this observation the viewer offers a feature to view only the tuple headers. All fields that are not tuples (we will denote these as *non-tuples*) are made invisible in this view. The result is a skeleton of hierarchical tuple headers. The structure of the configuration is now easier to see, and the key tuples in the configurations are easier to identify. This is especially useful for configurations that are cluttered with hundreds of fields. In figure 4.4.7.1 we see the result when we apply the skeleton view on the example from the previous chapter.

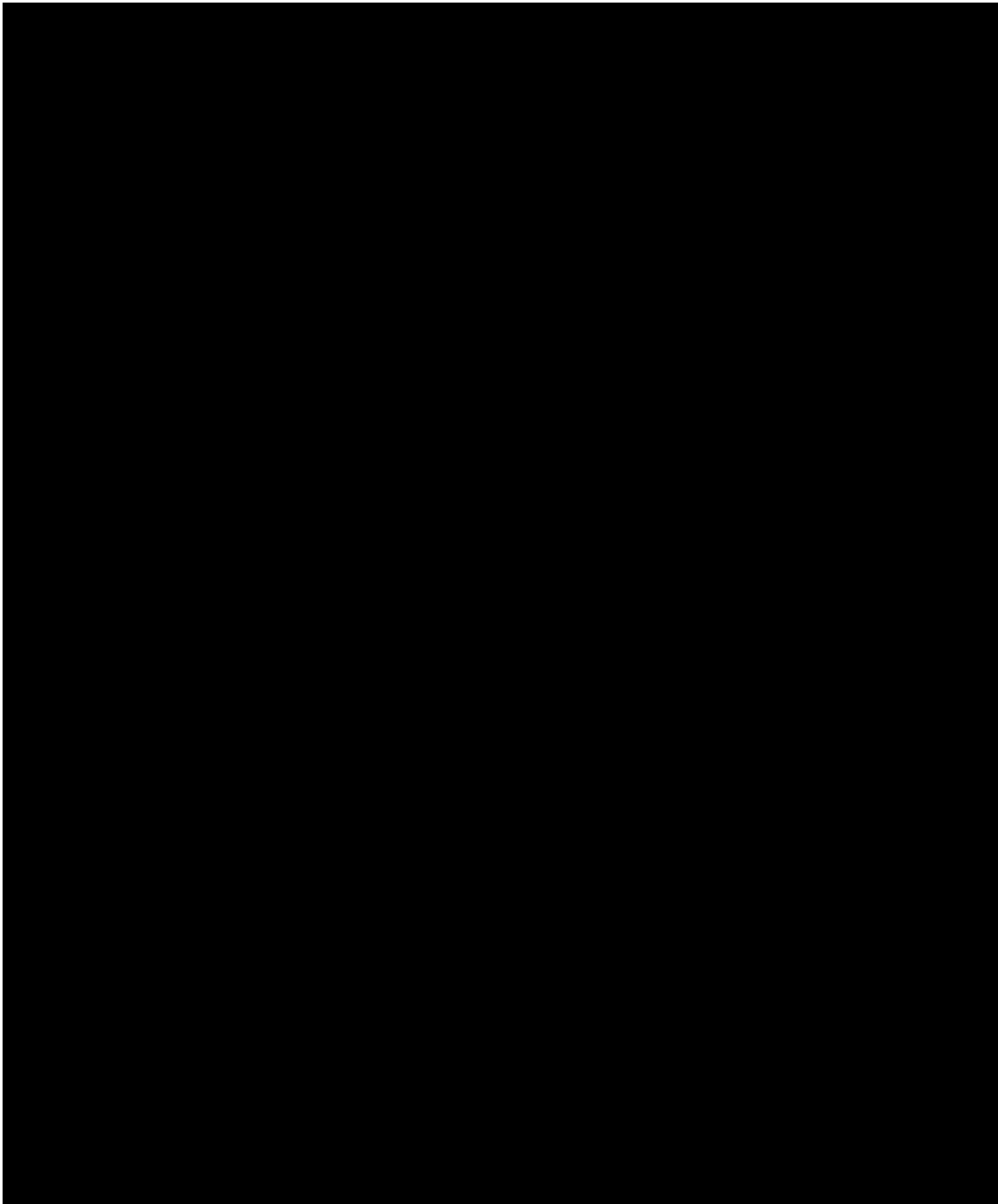


Figure 4.4.7.1: The Skeleton View

4.2.7.2 View Key tuples

We recall the notion of an *object*, which allows us to assign a type to a tuple. Objects are application specific. [REDACTED] These objects form usually the building blocks of a configuration. For that reason, they are a guideline for what the key tuples are in a configuration. Based on this observation, the viewer offers an *object list* in which all typed tuples are listed and grouped by their type. This makes it easy to determine the key tuples in a configuration. The viewer also offers the option to zoom on such a key tuple. Selecting a key tuple from the list narrows the view to that tuple, as we can see in figure 4.4.7.1.

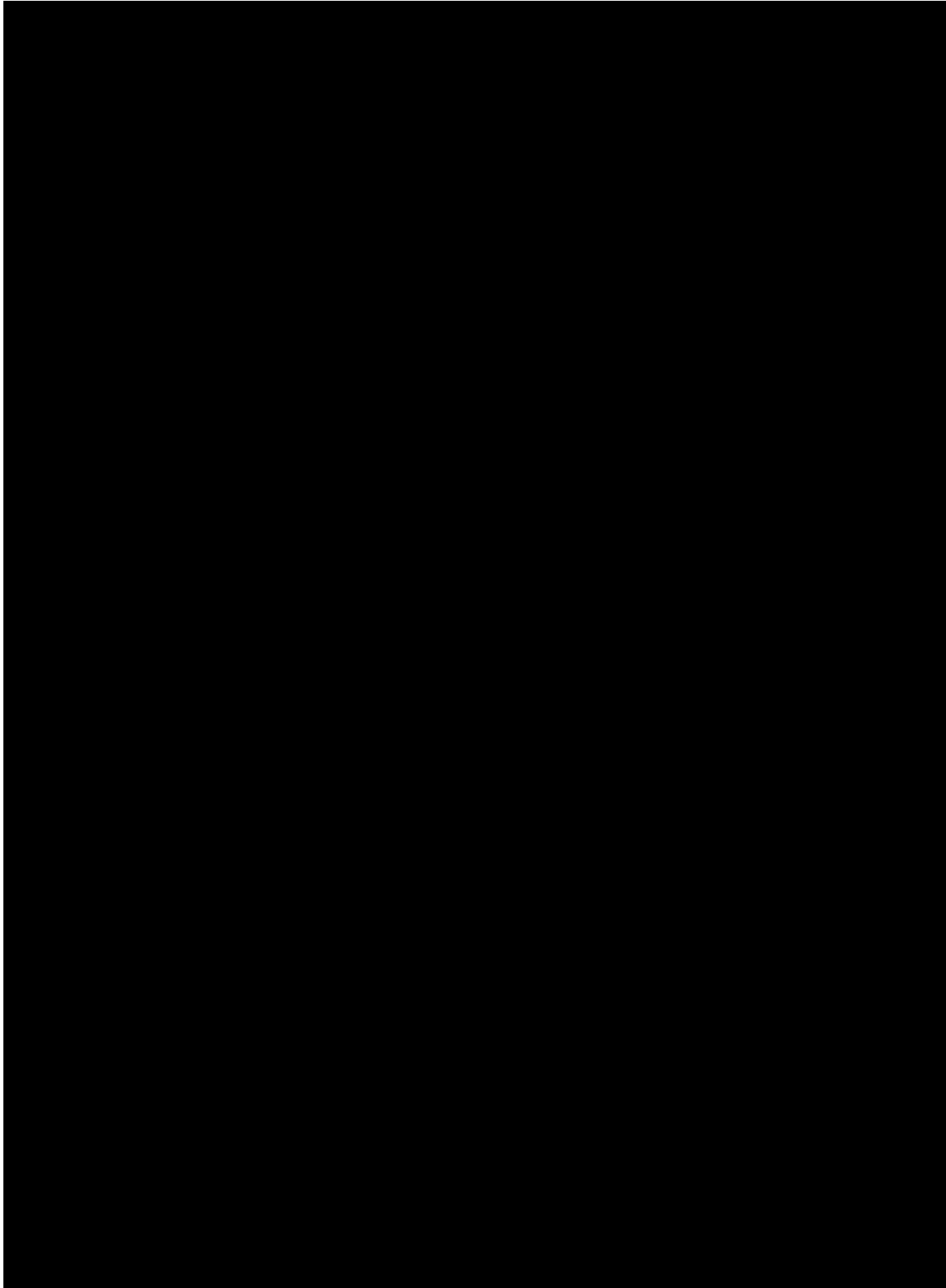


Figure 4.4.7.2: View Key Tuples

4.4.8 Browsing through a parameterized tuple

The viewer provides a feature that allows the user to see the collection of tuples generated from a parameterized tuple. To prevent the configuration from getting cluttered (in practice a collection can easily consist of hundreds of tuples), these tuples are not drawn one after the other. Instead, the viewer shows the first generated tuple by default, and allows the user to browse through the collection of tuples using a spin button. The viewer also integrates a drop down list in the header of the parameterized tuple (see figure 4.4.8). This list contains all the tuples that have been generated, identified by their signature (the list of parameters). The list allows the user

to quickly verify if the generated collection corresponds to what the user intended to define. The tuple currently being viewed can also be changed by selecting a tuple from the drop down list, as is illustrated in figure 4.4.8.

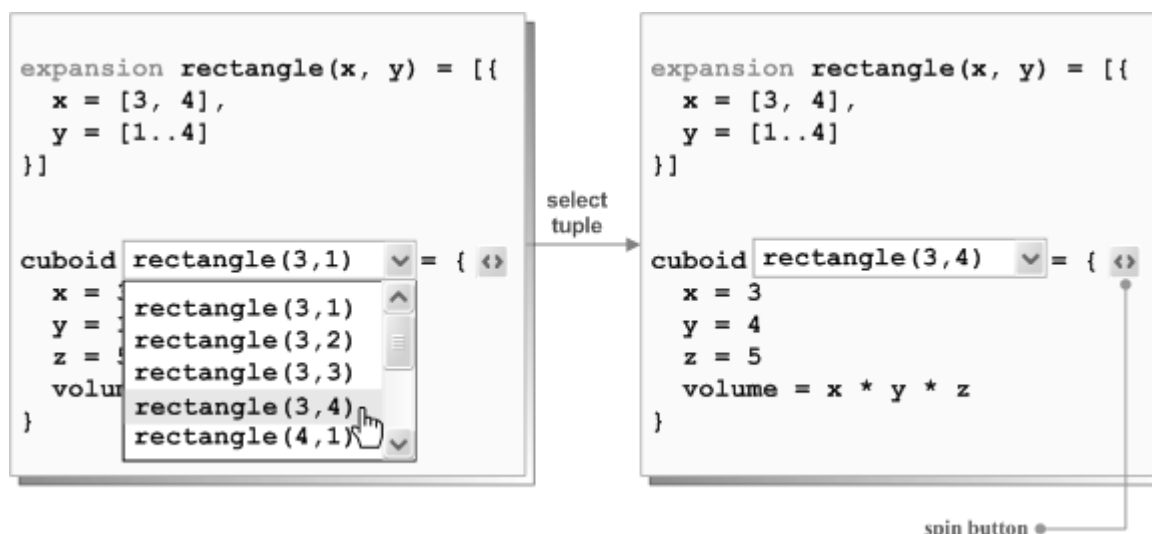


Figure 4.4.8: Browsing through a parameterized tuple

4.4.9 Navigating references

The viewer represents a GCL reference as a hyperlink to the referred location. This allows the user to easily determine which field is being referenced. When the hyperlink is clicked, the viewer navigates to the referred field, highlights this field, and centers it on the screen. An illustration of this is given in figure 4.4.9.

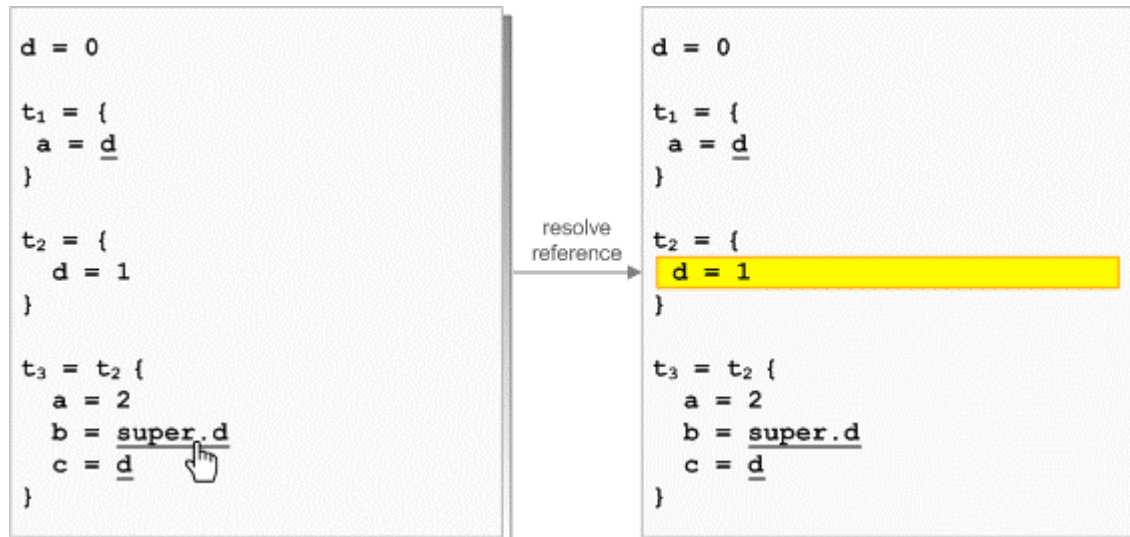


Figure 4.4.9: Navigating references

4.5 Additional Features

4.5.1. Enhancing the readability of configurations

Readability is a key factor in the process of understanding configurations. GCL Viewer uses some basic techniques to enhance the readability such as syntax coloring and code indentation. Syntax coloring is applied to help the user distinguish the different elements of GCL (e.g. tuple, expression, reference), while code indentation is used to improve the structure of the code. The

viewer also implements some more advanced visual techniques that aim to enhance the readability:

Foldable tuples

The viewer visualizes a tuple as a foldable container. Using a fold control that is provided in the tuple header the tuple can be collapsed and expanded (see figure 4.3.3a). When a tuple is collapsed, all its contained fields are made invisible. Folding tuples is useful when dealing with large configurations: Tuples that are irrelevant at that moment can be collapsed by the user, allowing the user to focus on a certain part of the configuration. Folding tuples can also be applied to enhance the overview in a configuration.

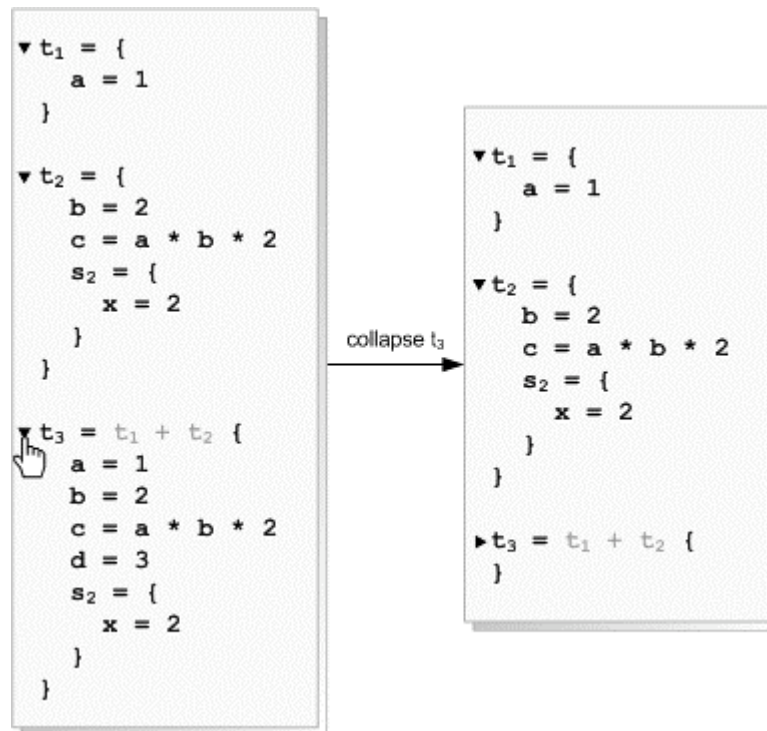


Figure 4.5.1a: Foldable tuples

Truncating long expressions

In practice expressions tend to become very long; they can easily span several lines of code, which can hamper the overview in the configuration. To improve the readability of a configuration, the viewer truncates an expression when it exceeds a certain amount of characters. The full expression is retrievable via a resize-button that is provided at the end of the truncated expression (see figure 4.3.3b).

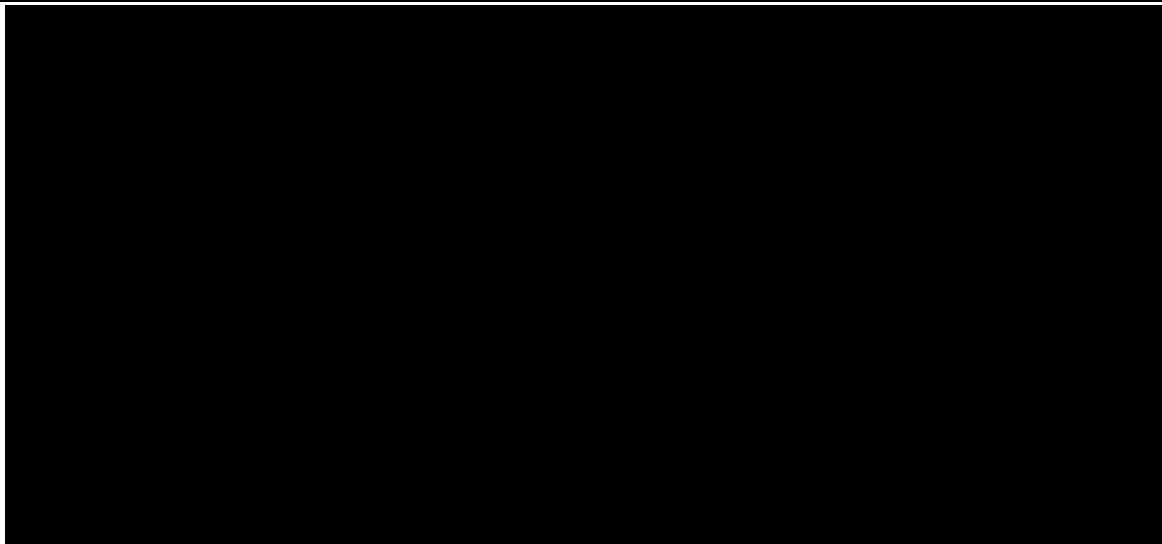


Figure 4.5.1b: Truncating expressions

Evaluating all expressions

In the example of figure 4.4.1b we have seen how evaluating expressions can lead to a more compact representation of a configuration (the evaluated form of an expression is usually much shorter than its original form). GCL Viewer offers the option to display all the expressions in the entire configuration in their evaluated form, enhancing the overview in the configuration. The original definition of the expression can still be retrieved by clicking on the evaluated expression.

Inherited an imported content

The expanded view (see section 4.4.2) displays a tuple including all of its inherited and imported content. In this view, GCL viewer uses the following visual techniques to distinguish between local, inherited, and imported fields: Local fields are displayed with a normal font, inherited fields are displayed as hyperlinks, and imported fields are displayed in italics and with a lighter font weight.

The motivation for using hyperlinks to indicate inherited fields is that they provide a convenient way to navigate to the location they were inherited from (the hyperlink links to the parent tuple of that field). We use the italics and a light font weight for imported content to indicate that these fields are not visible in the (original) definition view, but are projected by the tool to provide an explicit view.

4.5.2 Reporting evaluation errors and warnings

An expression can trigger error and/or warning messages when evaluated. We denote these messages as *evaluation messages*. The error or warning could have been caused by a fault in the expression itself or in a field referred by the expression.

GCL Viewer informs the user about the errors and warnings in a configuration. The viewer uses visual cues in the form of colored flags, placed in front of a field, to indicate that a field triggered an evaluation message(s) when its expression was evaluated. The color of the flag is assigned red if one or more of the messages is an error message, and otherwise orange. The evaluation message(s) can be retrieved by hovering over the flag, which pops up a panel showing for each error/warning the location of the error/warning and the error/warning message. In figure 4.5.2 we give an illustration of this.

foo.gcl

```
t = {
  x = 1
  y = 2
  z = cond(b, x, y)
```

Error: Expected literal of type boolean for argument 0 of cond()(found confBad)

Figure 4.5.2: Error and warning messages

The Viewer also provides an overview panel with all the evaluation messages in configuration.

4.5.3 Instant assertion result

We recall from chapter 2 that GCL offers two constructs for specifying assertions in a configuration. With the *assert* construct the user can specify a condition that emits an error when violated. The *expect* construct emits a warning instead of an error. The viewer provides an easy and intuitive way to see the result of these assertions. The viewer evaluates all the assertions in a configuration, and uses colored flags to indicate if the assertion failed or passed (see figure 4.5.3 for an illustration). We use the green color for an assertion that evaluates to true, and a red color for one that evaluates to false.

```
template triangle = {
  a = external,
  b = external,
  c = external
  assert c < a + b
}

isosceles = triangle {
  a = 1
  b = 1
  c = 1
}

bad_triangle = triangle {
  a = 1
  b = 1
  c = 3
}
```

```
template triangle = {
  a = external,
  b = external,
  c = external
  assert c < a + b
}

isosceles = triangle {
  a = 1
  b = 1
  c = 1
  pass : assert c < a + b
}

bad_triangle = triangle {
  a = 1
  b = 1
  c = 3
  fail : assert c < a + b
}
```

Figure 4.5.4: Instant assertion result

4.5.4 Field inspector

GCL Viewer offers the option to inspect the properties of a given field in the configuration. The *field inspector* is a panel that provides an overview of all the relevant properties of a field. The properties that are presented are:

- **Name:** The name of the field.
- **Selector:** A unique id of the field (which represents the path from the root to the field).

- **Type:** The type of field (this property is useful to GCL developers).
- **File name:** The name of the file where this field originates from.
- **Line number:** The line number where this field originates from.
- **Raw value:** The value of the field as was defined in the original expression.
- **Evaluated value:** The evaluated value of the field.
- **Inherited from:** The tuple from which the field was inherited (this property is blank when the field was locally defined).
- **Is Reference:** Records whether the field is a reference to another field.

The panel is drawn as a separate layer on top of the configuration. The panel is foldable and has also drag control. This allows the user to open several of these panels and move them next to each others, which is useful when comparing two (or more) fields. The field inspector is accessible via by holding down the CTRL key while clicking on the field to be inspected. In figure 4.5.4 we give an illustration of the field inspector. In this example the field inspector shown an overview of the properties of field $t_3.c$

Evaluated View

```

t1 = {
  a = 1
}

t2 = {
  b = 2
  c = a * b * 2
  s2 = {
    x = 2
  }
}

t3 = t1 + t2 {
  a = 1
  b = 2
  c = a * b * 2
  d = 3
  s2 = {
    x = 2
  }
}

```

t3.c	
name	c
selector	t3.c
ConfType	ConfBinary
filename	/home/user1/demo.gcl
line number	6
raw value	a * b * 2
evaluated value	4
inherited from	t2
is reference	false

Figure 4.5.4: The field inspector

4.5.5 Hybrid control

Next to control via a pointing device, GCL Viewer also full control via a keyboard. Using the keyboard we can browse and navigate through the code, and access all the features of the tool. GCL Viewer uses a cursor to indicate the current read position. Using the keyboard the cursor

can be moved to the next/previous field, or moved a level up/down (traversing nested tuples). Depending on the type of field at the cursor, we can use the keyboard to evaluate expressions, fold/unfold tuples, resolve references, display the inheritance info etc.

The motivation for the hybrid control is to accommodate both inexperienced users and advanced users. For a new user it is easier to use a pointing device to access the features. An experienced user can put more effort in remembering the keys and benefit from a fast and efficient way of using GCL Viewer.

4.6 Usability

In this section we reflect on the usability of our design of GCL Viewer. In figure 4.6 we show the usability quality attributes that we have defined.

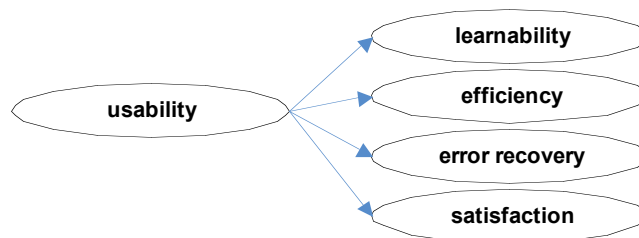


Figure 4.6: Usability quality attributes

Learnability

We promote easy learning of our tool by adhering to Shneiderman’s principle of *overview first, details on demand*. In the initial view of the tool we provide minimal additional information besides the source code. We also use visual cues in the form of icons, tooltips, and changing mouse cursors, to inform the users about the availability of more details and the availability of an interaction.

Efficiency

The hybrid control of GCL Viewer allows experienced users to use the keyboard to perform their tasks more quickly.

Error recovery

We have ensured that all the interactions of GCL Viewer can be undone. Users can therefore recover from unintended or accidental changes to the user interface. For example, clicking on an expression evaluates the expression, but clicking once more on it returns the original form.

Satisfaction

We have involved GCL users throughout the design process in order to improve our design according to the wishes of the users. However, a usability study needs to be carried out to determine the satisfaction of the users.

4.7 Summary

The main features of GCL Viewer were designed to address the key issues that were defined in the analysis described in chapter 3. In table 4.7 we show how the main features of GCL Viewer map to the key issues from chapter 3.

Key issue	Features
Complex expressions	Resolving expressions
Unclear definition of a configuration	Expanded View
Unknown origin of a variable	<ul style="list-style-type: none">▪ Origin of a field▪ Locating the origin of a field
Hidden dependencies	<ul style="list-style-type: none">▪ Dependency listing▪ Dependency coloring
Unintended modifications	Modification tags
Inheritance issues	<ul style="list-style-type: none">▪ Inheritance tags▪ Inheritance stack
Lack of overview in a configuration	<ul style="list-style-type: none">▪ Skeleton view▪ View key tuples
Erroneously defined parameterized tuples	Browsing through a parameterized tuple
Ambiguous references	Navigating references

Table 4.6 Features to key issue mapping

5. Technical design

In this chapter we describe the technical design of GCL Viewer. We start with formulating the design goals that have served as the guidelines for the design (section 5.1.1). Throughout the chapter we relate the design decisions that we made for our system to these design goals. We continue with another introductory section describing how a GCL configuration is processed by the viewer (section 5.1.2). In section 5.2 we present the major design decisions that form the fundament of the technical design of the viewer. We continue with describing the design of the viewer in a top-down manner, starting with a section on the system architecture (section 5.3), followed by detailed sections on each of the components of the system (sections 5.4 to 5.6). We conclude with section 5.7, summarizing the main aspects of the technical design.

5.1 Preliminaries

In this section we provide background information that is relevant for understanding the technical design that is presented in the remainder of this chapter. We define the design goals on which we have based our decision decisions, and we give insight how a GCL configuration is processed by the viewer.

5.1.1 Design goals

We have formulated our design goals as a set of quality attributes. These quality attributes form a guideline for the design of the system; throughout the chapter we will relate the major design decisions to these quality attributes. In figure 5.1.1 we present our quality attribute tree, providing an overview of the quality attributes that we have identified to be important for our design. In the remainder of this section we will describe the meaning of each quality attribute in the context of our project.

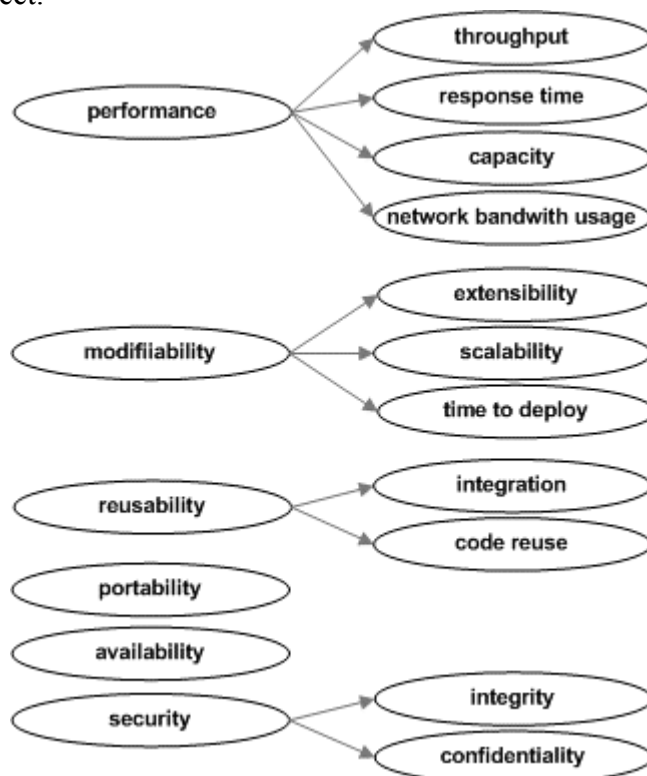


Figure 5.1.1: Quality attribute tree

Performance

We have learned from previous projects that processing GCL files is a computation intensive effort. Therefore, our design must aim to optimize:

- The time the user has to wait for a configuration to be ready to be shown (*response time*);
- The number of users that can be served in a time interval (*throughput*);
- The demand that can be placed on the system while continuing to meet responsiveness and throughput requirements (*capacity*);
- The demand that is placed on the network (*network bandwidth consumption*).

Modifiability

GCL is under continuous development; new features are being added regularly. Accordingly, our design must facilitate extending the viewer with new features (*extensibility*). Extensibility is also important because the viewer is developed in an iterative process; starting with a minimal product, and growing it steadily into a more advanced product. Our iterative development process leads to regular software updates and releases. It is therefore important to optimize the deployment of the software (*time to deploy*). Finally, since we expect both the user base and the size of configurations to grow over time, the system must be scalable in term of users and throughput (*scalability*).

Reusability

There are already some languages derived from GCL (e.g. ██████), and we expect this number to grow in the future. The design must allow the reuse of the tool for these languages (*reuse of code*). We also would like to make the analytical functionality of the viewer available to other tools. This means that subsystems of the viewer should also be designed as reusable components. Further, we have plans to integrate the viewer with online services, e.g. tutorials, wikis, and online monitoring systems (*integration*).

Portability

The user base of the viewer is known to have a variety of preferences for hardware and software platforms. The design of the system must aim to make it possible to run the viewer on different platforms / make the viewer platform independent.

Availability

The design must optimize the time that the system is up and running.

Security

The system must prevent GCL configuration from being altered (*integrity*). GCL configurations are confidential documents. The system must aim to prevent that these files are exposed to unauthorized users (*confidentiality*).

5.1.2 Processing a GCL file

To help understand some of our design decisions later on we will provide insight in how a GCL configuration file is processed before it can be displayed by the viewer. In figure 5.1.2 we present a pipeline defining the process tasks that a configuration file needs to undergo before it is ready to be displayed by the viewer. The process tasks will be described in the remainder of this section.

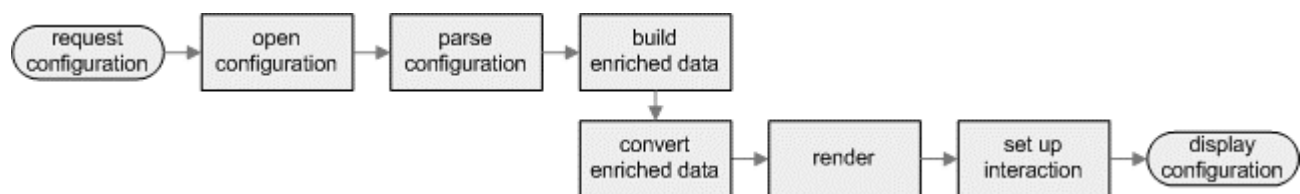


Figure 5.1.2: Flow chart showing the process tasks for a configuration

Step 1: Open and parse

The process is initiated by a request to view a configuration in the viewer. The configuration is then opened for reading and parsed first. The result of this stage is an internal data structure containing a tree representation of the configuration.

Step 2: Build the enriched data

In the next stage we traverse this tree and build the *enriched data*. The enriched data comprises all the information that is needed to render the configuration according to the specification from the previous chapter. To understand what data is compiled, we recall that a configuration is an ordered collection of attribute-value pairs first (we also refer to such a pair as a *field*). For each field, the enriched data records the attribute name, the value, and metadata describing additional properties of the field. In table 5.1.2 we give a full overview of the information that is compiled for each field in the configuration.

Data	Description	How obtained
selector	A unique id of the field describing the path from the root to the field	read out
type	The type of the field (e.g. tuple, literal, reference, assertion)	read out
attribute	The name of the field	read out
raw value	The original (unevaluated) value of the field	read out
evaluated value	The evaluated value of the field	computed
filename	The filename of the location of the field	read out
linenumber	The linenumber of the location of the field	read out
properties	GCL properties of the field (e.g. <i>local</i> , <i>final</i>)	read out
object type	The type of the tuple (in case the field is a tuple)	read out
parent	The tuple this field is inherited from	computed
modifies	The parent field that is being modified by this field	computed
referred filename	The filename of the referred field (in case the field is a reference)	computed
referred linenumber	The linenumber of the referred field (in case the field is a reference)	computed
subfields	The fields contained by the tuple (in case the field is a tuple)	computed

Table 5.2.2: The enriched data that is recorded for each field

The enriched data is obtained in two ways. Part of the enriched data is already available in the internal tree structure that was generated by the parser, and is obtained by simply reading out values. Examples are the attribute (name) and the type of a field. The other part of the enriched data is not readily available and needs to be computed on the fly. An example of such data is the inheritance information, e.g. the tuple a field was inherited from. Another example is the evaluated value of an expression. An overview of what data is read out and what is computed is provided in table 5.2.2.

The enriched data also encodes the structure of the configuration. Recall from chapter 2 that fields can be grouped and nested with the *tuple* construct. This hierarchy is encoded in the enriched data with the *subfields* property of a field (see table 5.2.2): In case the field is a tuple, we record in here the list of records describing the enriched data of the fields of that tuple. In figure 5.2.2 we give an illustration of the hierarchical composition of the enriched data. In this figure we see how a snippet of GCL code is represented in the enriched data.

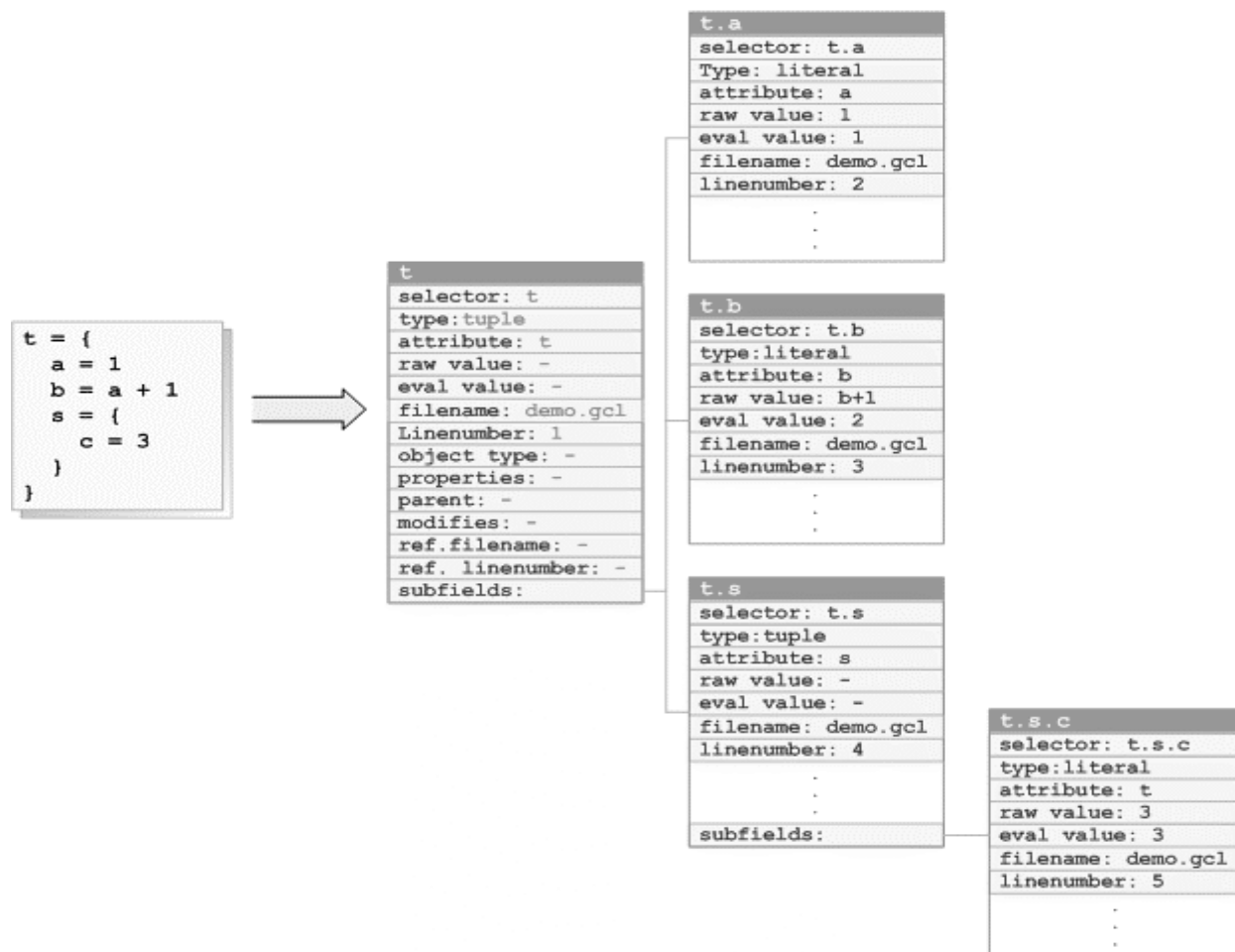


Figure 5.2.2: Hierarchical composition of the enriched data

Step 3: Convert the enriched data

The enriched data is stored in a platform independent format. This keeps the enriched data generic and allows us to use the enriched data for different purposes. A consequence of this is that the enriched data needs to be converted first to a format suiting the renderer before it can be rendered.

Step 4: Render and set up the interaction

In the next stage the enriched data is rendered into a visual representation. The renderer iterates over the structure and renders the fields one by one. Once the rendering has finished the interaction of the system is set up. In this stage event handlers are registered and initialized.

5.2 Design decisions

In this section we describe the major design decisions we made for GCL Viewer. For each design issue we compare design alternatives and motivation our choice. We use the quality attributes defined in section 5.1.1 as a guideline for comparing the design alternatives.

5.2.1 Distributed computing

From benchmark experiments we have learned that building the enriched data is a computational intensive and memory consuming task. Some configurations in production environment have shown to consume over 2 gigabytes of memory in this stage. Also the rendering stage is a computational intensive task. Taking this into account, we have decided not to design the viewer as a standalone application. This would lead to long response times and exhaust the resources

available on a typical desktop computer. Instead, we opt for a distributed approach in which we split up the system into parts (process tasks) and distribute these over several computers.

Another advantage of distributed computing is scalability. If the system would require more resources in the future, e.g. by extending it with new features, we can easily increase the number of computers to meet the increased resource demands. Moreover, since the different processing tasks can be distributed over different computers, we can scale the parts of the system independently, allowing us to fine tune the performance of the system (e.g. we can increase the number of computers that are assigned the task to build the enriched data).

A potential drawback of distributed computing is the overhead of network communications, which can negate the performance gain. But if we look at the communications in our system we observe two things. First, the data that is communicated is all textual data and relatively small in size. Second, since we have designed the system to send all the needed data at once rather than incrementally (see section 5.2.4), the number of communications between the program are optimized and kept at a low. These two things put a small load on the network.

Distributed computing implies a networked system, which leads to an increased risk . The recovery and load balancing mechanisms that are designed our system aim to and negative can have also a negative on the availability.

In figure 5.2.1 we present a table summarizing the trade offs for this design decision.

	responsiveness	scalability	Network bandwidth usage
standalone computing	--	-	0
distributed computing	++	+	-

Figure 5.2.1: Standalone vs. distributed computing

5.2.2 Web application

We choose to design the viewer as a web application. The main advantage being platform independence; the viewer is accessible from any computer with a working browser and requires no installation. Alternatively, we have considered designing the viewer as a desktop application, e.g. by integrating the viewer with an existing text editor. But the variety of platform and editor preferences among the users would have made it difficult to satisfy a majority of the users. Instead of writing different editor plug-ins, with a web application we can suffice to have one platform neutral application.

Another important advantage of a web application is that deployment and software updates come at a small cost. The application is downloaded automatically to the client by the browser. This frees us from deploying and installing new software at the client at every update, and fits our development process, in which we plan to add features to the viewer incrementally. The web application approach also conforms to our goal to make the viewer a read-only application. It provides us a sandboxed environment where files on the disk cannot be accessed or changed directly.

Our goal to integrate the viewer with (existing) online resources forms another important motivation for this design. The viewer can play a useful role in online documentation, tutorials and wikis on GCL. The tool would provide a different (enriched) view on the material, improving the understanding of it and making it interactive. A viewer that is designed as a web application can be seamlessly integrated with such online resources. Another domain of integration is remote monitoring. Some systems that use GCL for configuration purposes

provide online monitoring services to their users. There are already plans to integrate the viewer with these services. This would make it possible to have an enriched view on the configuration of the systems being monitored.

A drawback of a web application is the dependence on an internet connection. When an internet connection is absent, the system becomes unavailable. However, in the future we could extend GCL Viewer to support technology like Adobe Air [Air] or Google Gears [Gears], which would allow us to run the tool also offline.

In figure 5.2.2 we present a table summarizing the trade offs for this design decision.

	portability	deployment	integrity	integration	availability
desktop application	-	-	0	-	+
web application	+	++	+	+	-

Figure 5.2.2: desktop vs. web application

5.2.3 Rich Internet Application

Traditional web applications, also known as thin web clients, lack the rich interactivity and responsiveness that we know from desktop applications. HTML was not designed for building rich user interfaces, while the server round trip and the overhead of page-reload required upon each interaction lower the responsiveness of the traditional web application. Rich Internet Applications (RIAs), also known as rich web clients, aim to bridge the difference between traditional web applications and desktop applications. RIAs provide sophisticated user interfaces capable of representing complex processes and data, while minimizing client-server data transfers and moving the interaction and presentation layers from the server to the client [Bozzon06].

We have designed GCL Viewer as a Rich Client Application with the objective to benefit from the advantages of a web application (e.g. low deployment costs, portability) while maintaining a user experience that resembles desktop application closely. In the remainder of this section we discuss the RIA techniques that we have applied in GCL Viewer.

Client-side rendering

We chose to use HTML and JavaScript to implement our web application (we motivate this choice in section 5.6.2). A widely used technique in HTML based web applications is to render the HTML content at the server. In such a design, the client is a passive entity that requests its content from the server and displays this in the browser. The server serves the HTML statically (e.g. from a file system), or it generates the HTML dynamically using a template system (e.g. Clear Silver) or server-side scripts (e.g. JSP, Python, PHP). We refer to this technique as server-side rendering. An alternative technique is client-side rendering. In this approach, the client requests unformatted data from the server and renders this into HTML itself. The server sends a program along with the unformatted data, which is used by the client to parse the data and generate the HTML.

We have decided to apply the client-side rendering technique in our design. Specifically, we chose to use JavaScript, a client-side scripting language, to render the HTML for our application.

A main motivation for our choice is code modifiability; the client-side technique allows us to write code that is easier to change and extend. To understand this we need to note that a user interface has a visual definition (the layout) and a behavioral definition (the interaction). We use HTML to specify the visual definition of the user interface. The behavior (e.g. detect user

actions and react on them), however, cannot be specified with HTML; a browser language such as JavaScript must be used for that purpose. This means that in the case where the HTML is rendered at the server, the user interface code would become split over two applications (the application rendering the HTML and the JavaScript application providing the interaction), which makes the code more difficult to write and maintain. With our client-side approach we can write a self-contained JavaScript application that takes care of all the aspects of the user interface. The client application is not dependent on some functionality (e.g. rendering) deployed at the frontend server. This way we also localize modifications and decrease the coupling between the client and the server application. Restricting modifications to a small set of modules will generally reduce the cost [Bass].

Another advantage of the client-side rendering technique is that it puts a smaller load on the network compared to the server-side rendering. The rendered data is several orders of magnitude larger in size than the enriched data. In the client-side technique we only send the enriched data plus the JavaScript application file to the client, whereas in the server-side technique we send the much larger rendered data.

Because the rendering task is moved to the client, the computational load for the server also decreases, leading to an increased throughput of the server. Finally, the decreased footprint for the server also increases the scalability of the server.

Rich user interface

Client-side scripts such as JavaScript allow us to implement rich user interface widgets which are not standard available in HTML (e.g. color picker, drag-drop panel, foldable widget). Client side scripts also offer more flexibility and control in working with HTML compared to server-side scripting. This is because client-side scripts can access and manipulate the HTML code via the Document Object Model (DOM) that is provided by the browser. The DOM provides an internal object representation of the HTML document (as opposed to its textual representation), which can be used to access and manipulate the HTML in an object-oriented way. This also allows us to implement some of the features of GCL Viewer more efficiently (e.g. attaching event handlers to rendered HTML elements can be done on the fly), while the object oriented approach leads to better maintainable code.

Asynchronous communications

In the traditional web application model user actions in the user interface trigger a request to the server. The server then performs some computations, retrieves data and does some processing, and returns the result to the client. The client reloads the page to display the new content. Throughout the server roundtrip the user interface was blocked and the user had to wait until the new page was loaded. The drawback is that it yields a user experience that is different from desktop applications where applications are more responsive. To improve this, we have decided to use AJAX [Garrett05] technology in our web application to provide *asynchronous* communications between the client and server. The client application does not block to wait for the data to arrive; instead it specifies a call back function that is called when the data arrives. This approach improves the responsiveness of the interaction of our application, and leads to an improved user experience.

Network efficiency

In a traditional web application, when an action in the user interface requires new data, a roundtrip to the server is made to request this data, which can lead to a bad responsiveness of the user interface. In order to improve the overall responsiveness of the user interface, we have decided to send all the enriched data in one go to the client, rather than on demand. For instance, the evaluated values of expressions (section 4.4.1) are sent to the client initially with the

enriched data. When the user clicks to evaluate an expression, the data (in this case, the evaluated value of the expression) is already at the client, saving us the roundtrip to the server, which promotes the response time of these interactions.

A drawback of this approach is that it leads to a longer initial load time of the user interface (because the amount of data that needs to be sent initially from the server to the client is larger). However, this sacrifice is compensated with an overall increase in responsiveness: Once the user interface has been loaded, the interactions can be handled locally and do not require a server roundtrip to the server. Another advantage of this is that the server does not need to maintain a communication state, which simplifies the design of the server. Each request is an independent transaction that is unrelated to any previous request. Finally, it leads to a reduced number of network communications overall, since the client applications do not need to request data upon each action in the user interface.

Based on our observation that computing the collection of tuples generated by a parameterized tuple is a very resource intensive task, we have decided to design a different mechanism to handle interactions on parameterized tuples. Instead of sending all the tuples generated by the parameterized tuple initially, we send them on demand. Initially we send a list containing the headers of the generated tuple instances. Upon a selection of a tuple instance (see figure 4.4.8), we send an asynchronous request to the server for the enriched data for the tuple instance. We note that parameterized tuples is an advanced language construct which does not occur in GCL configurations as often as expressions do, for example. The number of these type of communications are therefore relatively low.

In figure 5.2.3 we present summarize the discussion on the RIA techniques by comparing our RIA approach to a traditional web application.

	extensibility	code reuse	initial response time	overall response time	network usage	throughput	scalability
RIA	++	++	-	++	+	+	+
traditional web application	--	-	+	-	-	-	-

Figure 5.2.3 Rich Internet Application vs. traditional web application

5.3 Architectural design

In this section we describe the architectural design of the viewer. In section 5.3.1 we present the system architecture, and introduce the main components of our system. In section 5.3.2 we describe the physical deployment of these components, and in section 5.3.3 we specify the data interchange between the components. In section 5.3.4 we give an operational view on the system, and we conclude with the fault tolerance aspects of the system in section 5.3.5.

5.3.1 System architecture

We have designed our system according to a client-server architecture. Figure 5.3.1a shows the system architecture and introduces the three main components of GCL Viewer: *client application*, *frontend service*, and *backend service*. The client application provides the user interface and runs at the client. The frontend and backend services run on the server and are used to perform most of the processing tasks (the exact task division between the components will be explained further on).

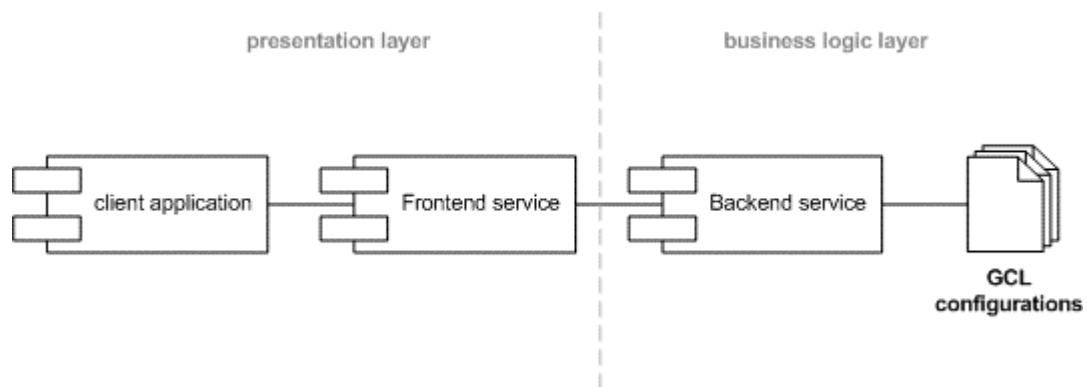


Figure 5.3.1a: The system architecture of GCL Viewer

Next to distributing the computational load, the client-server architecture allows us to enforce data integrity. The configuration files are accessible by the backend service only (see figure 5.3.1a). The backend service only sends a view on the configuration to the client without sending the actual configuration. This prevents the configuration from becoming altered (accidentally) by the client.

Presentation and business logic layer

An important architectural design decision is to enforce a separation between presentation and business logic code in our system. This allows us to deal with the different aspects of the problem in isolation. To that purpose we have defined two layers in our system; a presentation and a business logic layer (see figure 5.3.1a). In the presentation layer we deal with presentation aspects (e.g. rendering the user interface), while the business logic layer concerns itself with computational aspects of the problem (e.g. parsing the configuration, computing the enriched data). The separation allows us to design independent components. The presentation layer does not need to worry how the data that it needs for rendering was computed. And likewise, the business logic layer does not need to concern how the computed data is going to be presented. The advantage of the separation is that the layers can be developed and tested in isolation, without bringing in the complexity of other layers. Moreover, the system becomes easier to modify, because (most) changes become confined to one layer. For instance, we can change the way the data is presented without needing to alter the business logic layer.

Task distribution

In figure 5.3.1b we show how the processing tasks that we defined in section 5.1.2 are distributed over the components.

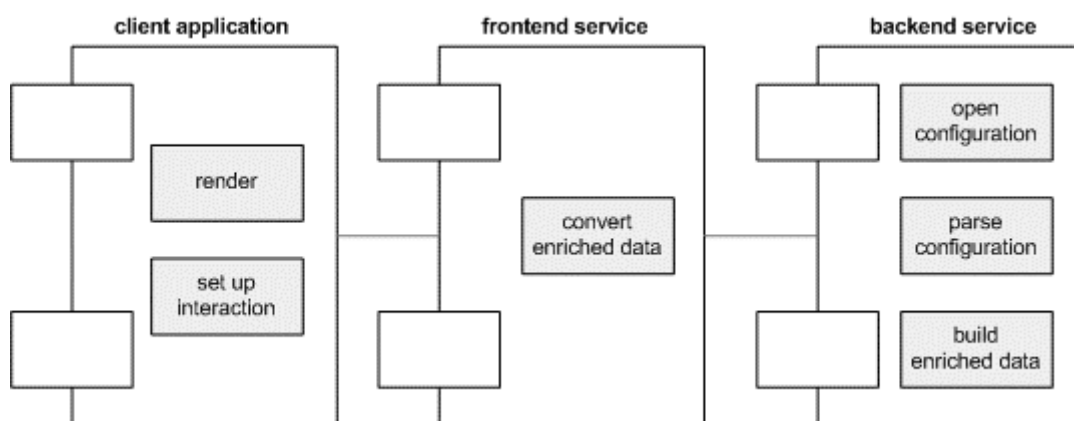


Figure 5.3.1b: The distribution of the processing task over the main components

The backend service is mainly concerned with preparing the enriched data needed for the rendering stage. The client application renders the user interface containing a visualization of the configuration, and implements the interaction of the tool. The frontend service has been introduced to maintain the separation between presentation and business logic code throughout the system: server tasks that have a presentation aspect are dealt with in the frontend service. In a design that applies the server-side rendering technique, the rendering of HTML would take place in the frontend service component. In our client-side rendering approach, the rendering stage takes place at the client; hence our frontend service has a relatively small task. It is assigned the task to convert the enriched data to a format that suits the client application (JSON). To promote reuse of our system, it is important to maintain the separation of a frontend and backend service. If we want to switch the system to a server-side rendering approach, for example, we only need to replace the frontend service. The backend service would remain the same then. Moreover, the frontend service serves also other purposes; it acts as a web server to the client, it sanitizes client requests, and it implements a load balancing mechanism, which distributes the client requests over several instances of the backend service. In section 5.5 we will elaborate more on these tasks.

5.3.2 Physical view

In figure 5.3.2a we present a physical view on our system, depicting how the components are deployed on computers.

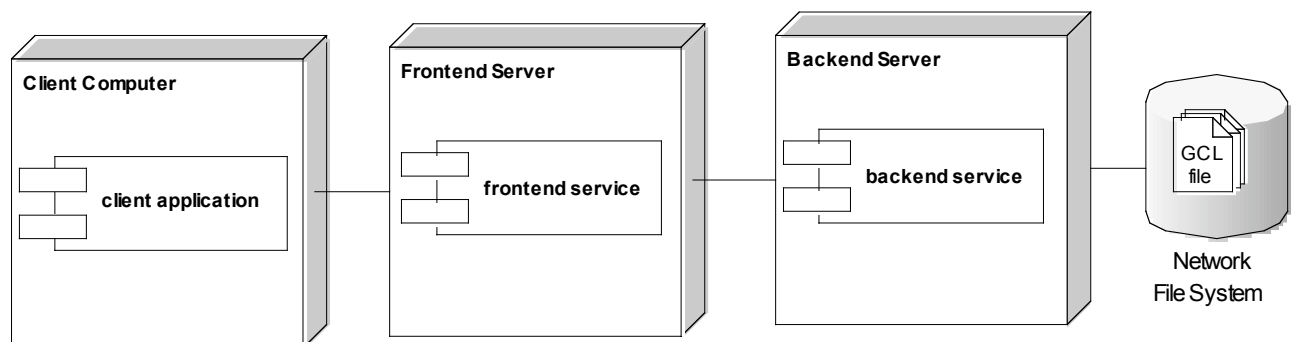


Figure 5.3.2a: Physical view

We have designed the frontend and backend service as separate applications. This allows us to deploy these applications on different computers and run them independent of each others (see figure 5.3.2a). The partition is transparent to the client. The client is only aware of the existence of one server, the frontend server, and requests the data it needs from this server.

The physical separation brings several advantages. First, it provides us an additional level of distributing the computation load: The business logic code and presentation code can be run on different computers, each with their own resources. We can also stop and (re)start the two services independently, which is useful when we want to deploy changes that are limited to one service only. Another advantage is that we can scale the resources for the business logic work and the presentation work independently, allowing us to fine tune the performance of the system. For instance, if computing the enriched data is a performance bottleneck, we can increase the number of machines that run the backend service (while keeping the number of frontend machines constant). This gives us more freedom to adapt the system to accommodate future changes in the number of users or in the resource consumption of GCL processing. And finally, this approach also promotes reuse. As we will see in section 5.4, the backend provides functionality that is common for analyzing GCL configurations (i.e. not specific for GCL Viewer), and could therefore also be of interest to other (analytical) applications. We could, for

instance, have another tool that uses our backend to acquire the data or functionality it needs. Or we could have a plug-in for an editor that uses the same functionality as the viewer. Figure 5.3.2b illustrates a scenario in which different frontend services are connected to one backend service.

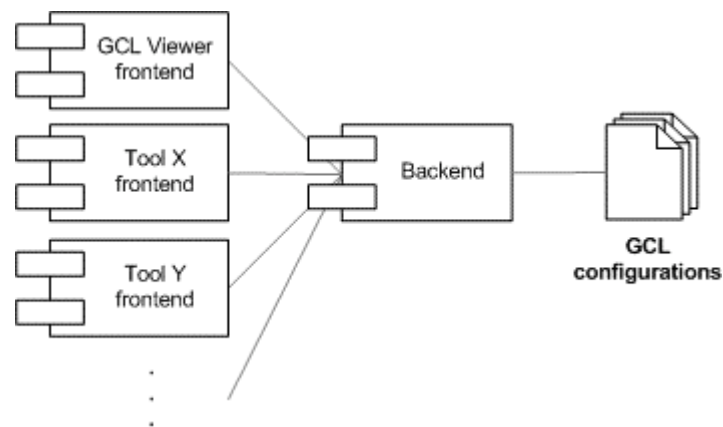


Figure 5.3.2b: Connecting different frontend services to one backend service

In a similar way we could reuse our frontend for some other backend that generates enriched data for another (configuration) language. This is useful if we want to reuse the presentation code for another application, e.g. to visualize some other (configuration) language. In that case we have to ensure that the new backend provides its data in a similar format as was defined in section 5.1.2.

5.3.3 Data interchange format

In figure 5.3.3a we provide an overview of the data interchange formats that are used for communications between the components of our system. We use the Protocol Buffers format [Pike] for the data interchange between the frontend and backend service. Protocol Buffers is a platform independent data format developed at Google. A protocol buffer is a collection of data records that can be converted into native C++, Java or Python data holder classes, including code to access the fields. Protocol Buffers can also be serialized over the network, for example as part of a Remote Procedure Call. An advantage of the platform independence of protocol buffers is that it allows us to design the backend as an open service: the backend becomes a general purpose service that can be utilized by other (analytical) applications. Protocol buffers provide also good flexibility; they allow us to specify specialized data for the viewer, and mark this data as *optional* fields in the records, meaning that it can be ignored by other applications.

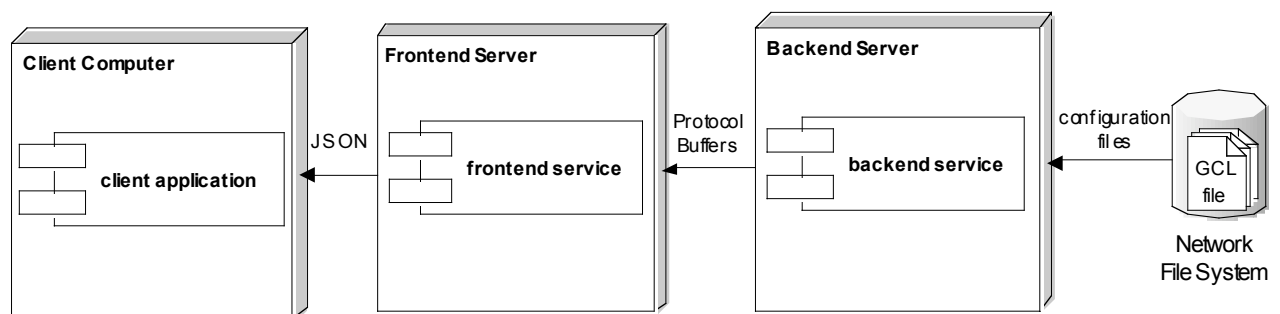


Figure 5.3.3a: Data interchange formats

We use the JavaScript Object Notation (JSON) as the data interchange format between the client application and the frontend service. The motivation for this is that the JSON format

accommodates our client application, which is implemented in JavaScript. JSON is a subset of the object literal notation of JavaScript; JSON data are essentially serialized JavaScript objects. As a result it is fast to import and easy to use in a JavaScript application. JSON has also a compact notation (compared to XML, for example), which has a positive influence on the network bandwidth usage of our system. In figure 5.3.3b we give an example that compares JSON with XML.

JSON	XML
<pre>{ "field": { "name": "x", "value": "y+1", "filename": "/home/user/demo.gcl", "linenumber": "15" } }</pre>	<pre><field> <name>x</name> <value>y+1</value> <filename>/home/user/demo.gcl</filename> <linenumber>15</linenumber> </field></pre>

Figure 5.3.3b: JSON vs. XML

To promote the reuse of our system, we allow the data interchange format between the frontend service and the client application to be easily changed. When the system is (re)used for a different type of client application, the data format can be changed to accommodate that application. In section 5.5.2 we will explain how we can use different frontend renderers to vary the output format of the frontend.

5.3.4 Operational view

In this section we use sequence diagrams to illustrate how the components in our system interact with each other.

In figure 5.3.4a we show the communications between the components that take place when a user requests to view a configuration. The process is initiated by a request from the browser for the main page of GCL Viewer containing the client application. The main page is served and returned by the frontend service. A user can now request to view a configuration file in GCL Viewer. This results in a *view file* request from the client application to the frontend service. This request is asynchronous; the client application does not block to wait for the response. Next, the frontend service sends an asynchronous request to the backend service for the enriched data of the requested configuration file. Once the backend service has computed the enriched data it returns the call to the frontend service, which then converts the enriched data to the JSON format, and finally returns the call to the client application.

In case of a configuration file that contains parameterized tuples (section 2.2.7), the user can also request to view the next tuple instance in the collection of tuples that was generated by the parameterized tuple. This triggers a *view tuple* asynchronous request to the frontend service (figure 5.3.4a). The frontend service request the enriched data for the tuple at the backend service and returns this to the client.

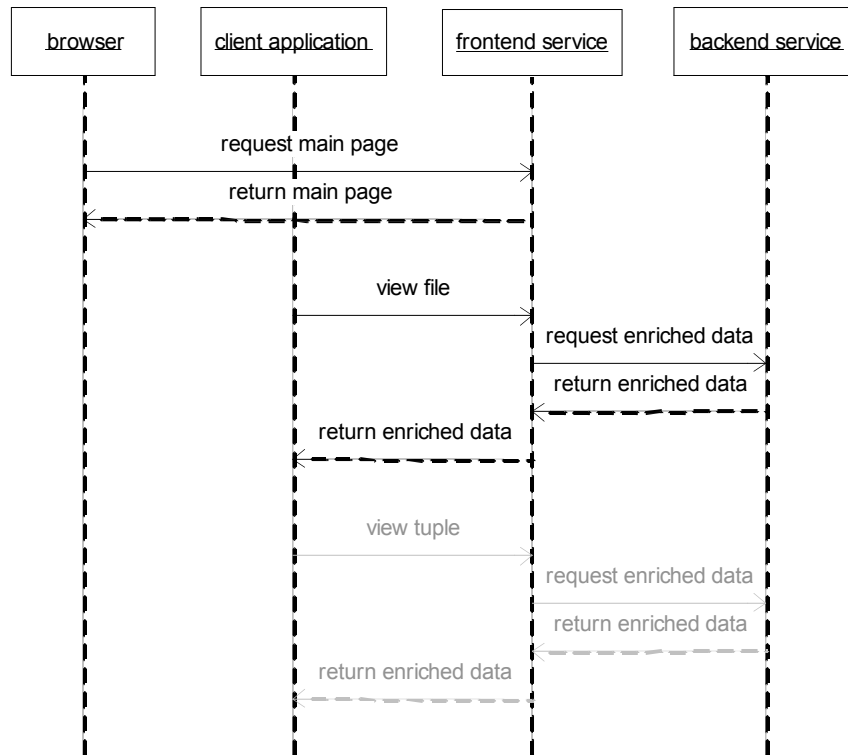


Figure 5.3.4a: The user requests to view a configuration file

In figure 5.3.4b we illustrate the interactions between the components in a more complex scenario consisting of two instances of the client application, one frontend service instance, and two instances of the backend service. In this scenario we illustrate how the asynchronous communications allow the frontend service to handle multiple requests, and how the frontend service balances the load over a pool of backend services.

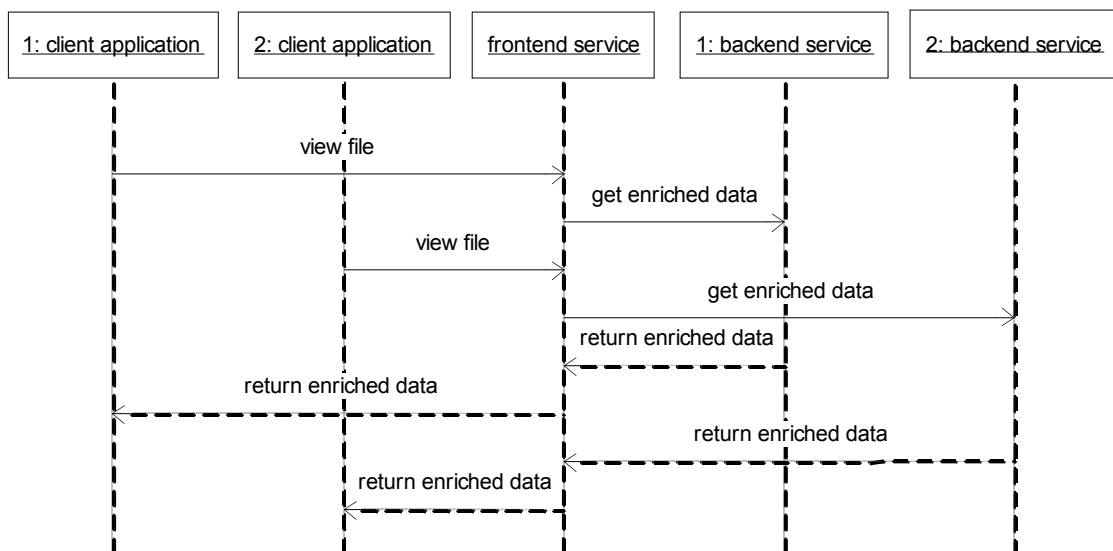


Figure 5.3.4b: The frontend service balances the requests of several client applications over two backend service instances

The frontend service receives a request to view a configuration file from client application 1. It sends an (asynchronous) request for the enriched data to backend service 1. Before the result of this request is returned, the frontend service receives a request to view a configuration file from client application 2. We assume now that in this scenario the load balancing algorithm of the frontend service has decided that subsequent requests should be send to backend service 2. Therefore, the frontend service sends a request for the enriched data (for the configuration file requested by client application 2) to backend service 2. In the mean time backend service 1 has finished computing the requested enriched data, and returns this to the frontend service. The frontend service converts the result to JSON and sends it to client application 1. And finally, once backend service 2 has finished computing the enriched data, it is forwarded in similar way to the client application 2.

5.3.5 Fault tolerance

GCL Viewer is designed to recover from component failures. We have implemented a recovery mechanism that automatically restarts a backend or frontend service in case it crashes or fails to respond.

After we have deployed GCL Viewer we learned that most of the failures occur in the backend service. To improve the availability of our system, we run several instances of the backend service. The frontend service is assigned a pool of backend services and switches between them depending on the availability and the load executed on the backend services. A backend service that is restarted after a failure automatically rejoins the pool.

The separation into a frontend and backend service allows us to provide a better service to the client in case of failures: If the backend has failed or is unavailable we can still serve a web page from the frontend and include a message informing the user about the problem.

5.4 Backend service

In this section we describe the backend component of GCL Viewer. We explain the purpose of the backend component, provide an overview of the method and techniques that we have used, and give insight in the design of the backend component.

5.4.1 Purpose

The main task of the backend is to compile and provide the enriched data for a given configuration. The backend provides the enriched data over the network via Remote Procedure Calls (RPC). The backend also implements general analytical functionality for GCL such as methods for evaluating expressions in a configuration and methods for computing the collection of tuples generated from a parameterized tuple. An overview of the functionality provided by the backend is given in section 5.4.3. Our aim is to make this functionality also available to other GCL tools. To that purpose, the backend is designed as a server that exposes this functionality over the network. An application that is interested in using this functionality can connect to the backend and request the functionality it wants.

5.4.2 Method and techniques

Before the backend can perform computations on GCL configurations, e.g. compile the enriched data, it will need to parse and evaluate the configuration first. Instead of implementing our own tools for this, we have decided to reuse the resources available in the GCL Library (e.g. parser,

evaluator). This speeds up the development of the backend, but also ensures that changes and extensions to GCL are immediately reflected in our tool.

The GCL Library is an extensive software library. Much of the functionality in there is not relevant to analytical tools such as GCL Viewer. Therefore we have decided to put an abstraction layer on top of the GCL library. This is implemented in the form of a class which we denote as *GCL Analyzer*. The purpose of this layer is to hide functionality from the GCL Library that is not needed, and to implement and provide specialized functionality that is useful for analytical tools such as the viewer.

GCL Analyzer implements a variety of functions for analyzing configurations, e.g. evaluate expressions, trace the inheritance of a field, and compute the tuples generated by a parameterized tuple. GCL Analyzer is designed as an interface class that can be sub-classed to create specialized analyzers for languages derived from GCL.

This idea is illustrated with the UML diagram in figure 5.4.2.

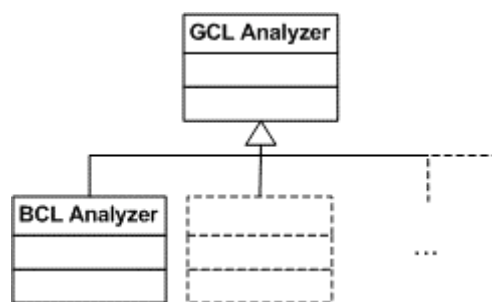


Figure 5.4.2: Specializations of the GCL Analyzer class

The backend also implements an RPC server. The purpose of this server is to make the functionality from the GCL Analyzer available to applications across the network. The data communicated by the RPC Server is in the Protocol Buffer format (see section 5.3.4), ensuring platform independence of the backend. The RPC server implements a concurrency mechanism that allows several applications to connect at the same time. For each request the server creates an instance of GCL Analyzer. The instance is cleared and released at the end of the request, maintaining a stateless server.

5.4.3 Detailed design

In figure 5.4.3a we present a UML diagram depicting the main classes of the backend.

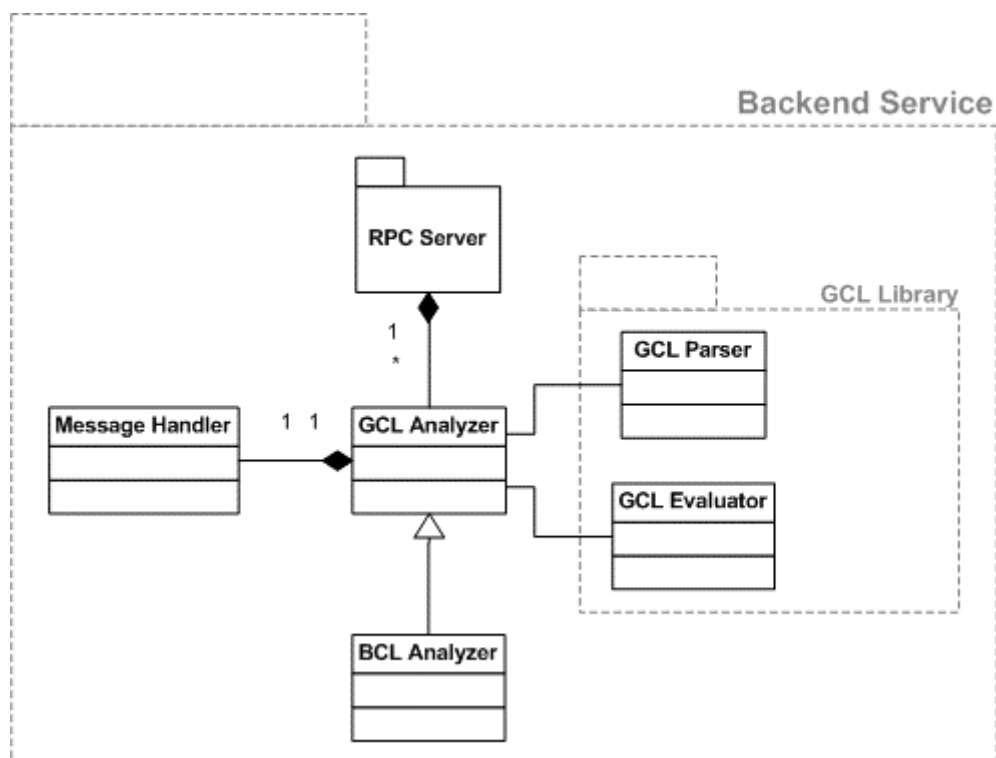
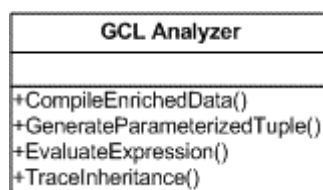


Figure 5.4.3a: The main classes of the backend

GCL Analyzer

This class implements the analytical functionality that is provided by the backend. In the context of GCL Viewer, the key task of this class is to compute the enriched data of a configuration. To that purpose it parses the requested configuration, and performs analysis and computation on the configuration to generate the necessary data, e.g. the inheritance trace of a field, the evaluated values of expressions, and errors or warnings triggered by erroneously defined expressions. GCL Analyzer also provides this functionality independently from the enriched data; the class offers a set of methods that can be used by other applications to analyze configurations. In figure 5.4.3b we give an overview of the methods exposed by GCL Analyzer. The goal is to extend this class with new methods in the future, providing more analytical functionality to GCL tools.



5.4.3b Methods exposed by GCL Analyzer

GCL Parser

This class is used to parse a GCL configuration. The result of parsing a GCL configuration is an internal tree structure representing the configuration

GCL Evaluator

This class is used to evaluate a GCL configuration (e.g. evaluate expressions, resolve tuple inheritance).

Message Handler

This class is used to collect the error and warning messages that are triggered by the evaluator.

RPC Server

This class provides the functionality of GCL Analyzer over the network.

5.5 Frontend service

In this section we describe the frontend service component of GCL Viewer. We explain the purpose of the frontend service (section 5.5.1), provide an overview of the method and techniques that we have used for it (section 5.5.2), and present a detailed design of this component (5.5.3).

5.5.1 Purpose

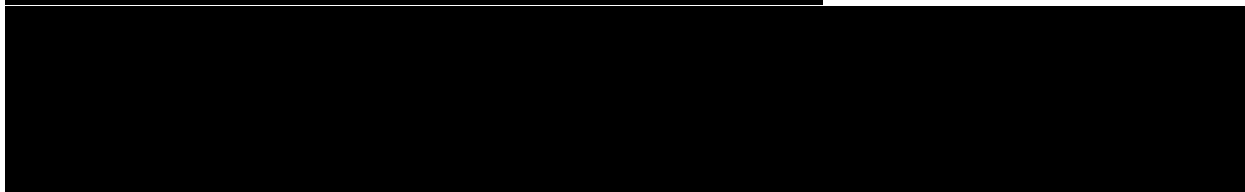
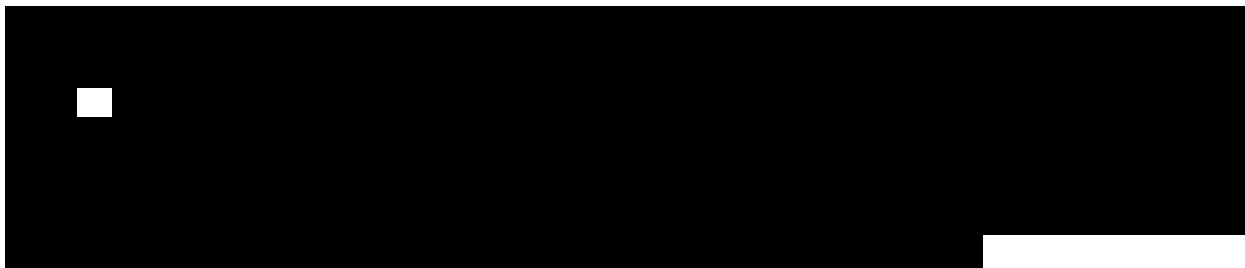
The frontend service prepares the enriched data for presentation by the client. The meaning of ‘prepare’ depends on the architecture style of the system. In an architecture that applies the server-side rendering technique, the frontend is assigned the task to render the enriched data into a presentation format such as HTML. In our design we chose to let the rendering stage takes place at the client (the client-side rendering technique). We have assigned the frontend the task to convert the enriched data that is generated by the backend into the JSON format (JavaScript Object Notation).

The frontend service is the communication peer of the client application. The client application requests the data it needs from the frontend. It is the responsibility of the frontend to handle these requests and serve the requested data. Depending on the type of the request, the frontend either serves the result data by itself or it requests data from the backend first and uses this to generate the result. An example of the first is a request for static content, such as the initial web page containing the client application. An example of the second type is a request to view a GCL configuration. For this request, the frontend requests the enriched data from the backend first, and then converts this to JSON before sending the result back to the client application.

The choice for a web system implies that the frontend also acts as a web server to the client. The frontend therefore supports communication via the Hypertext Transfer Protocol (HTTP).

Another purpose of the frontend service is to provide load balancing by distributing the client requests over a pool of backend instances.

5.5.2 Method and techniques



[Redacted text block]

[Redacted text block]

- [Redacted list item]

- [Redacted list item]

- [Redacted list item]

[Redacted text block]

[Redacted text block]

Figure 5.5.2: [Redacted caption]

[Redacted text block]

5.5.3 Detailed design

[Redacted text block]



Figure 5.3.3: Class diagram of the frontend

[Redacted text block]

[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]

5.6 Client application

In this section we describe the design of the client application. The client application provides the user interface of GCL Viewer. In section 5.6.1 we describe the purpose of the client application, in section 5.6.2 we specify the methods and techniques that we used in the client application, and in section 5.6.3 we give a detailed design of the client application.

5.6.1 Purpose

The client application provides the user interface of GCL Viewer. It renders a GCL configuration according to the visual design that is specified in chapter 4. The client application is also responsible for providing the interactive behavior of the user interface described in that specification. The client application is served by the frontend server and is send along with the initial web page that is sent to the client when GCL Viewer is requested from the browser.

5.6.2 Method and techniques

HTML and JavaScript

We recall our design decision to implement GCL Viewer as a Rich Internet Application (section 5.2.3). We use HTML to implement the visual aspects of the user interface, and JavaScript to add dynamic elements to the user interface (e.g. change the visibility or color of text), and for providing the interactive behavior of the application (e.g. respond to mouse and keyboard actions).

We chose for HTML and JavaScript because both languages are supported by most common browsers and do not require the installation of plug-ins. Moreover, HTML is an effective language for describing text based information in a document. This makes it very suitable for implementing our user interface, which is in essence a document containing formatted text (representing GCL source code). HTML is a static markup language; we use JavaScript to implement the interactivity of the user interface. Other client-side scripting languages (e.g. VBScript) could have been used for the same purpose, but JavaScript is the most widely supported in browsers and provides powerful functionality to work with HTML. Moreover, the

choice for JavaScript allows us to reuse code from Google’s extensive JavaScript Library (and speed up the development of the client application), and contribute new code to this library.

A self-contained JavaScript component

In contrast to traditional web applications that enrich HTML code with JavaScript code fragments in order to add interactive functionality to the web page, we chose to design the viewer as a self-contained JavaScript application. The JavaScript application implements all the functionality of the client application, including the rendering of the HTML, and does not depend on functionality in other client scripts or on code running on the web server. This way the client application becomes a self contained software component that can be easily reused in other web services. We refer to section 5.2.3 for more advantages of this approach.

An object-oriented approach

Although JavaScript is a prototype-based language rather than an object-oriented language, it is possible to write an application in JavaScript according to the object-oriented paradigm [MDC]. Specifically, we can write an application in JavaScript that makes use of the main concepts of object-oriented programming, being inheritance, encapsulation, and polymorphism. The advantage of the object-oriented approach is that it promotes the modifiability and reusability of the client application. It allows us to design a modular application, provide services while hiding implementation details, and create specializations of the client application

There are different techniques around of how object-oriented programming can be applied in JavaScript. We have chosen to comply with the technique that is used in Google’s JavaScript library. This leads to consistent code, and promotes the reuse of our application within Google. We will give a brief description of this technique in the remainder of this section.

A JavaScript based template system

Template systems are widely used by web applications for rendering web documents. Their main advantage is that they provide a convenient way to separate the presentation code (HTML) from the application code: The HTML code is written and maintained in template files outside the application code. This also allows us to write the formatting in HTML style instead of a collection of concatenated strings, which is usually the case when HTML is written from within the application code. Next to making the code easier to maintain, template systems also promote reuse (a template can be reused for different (parts of) web documents), and provide an isolation of changes (e.g. we can change the appearance of a (HTML based) user interface without touching the application logic).

A template also provides a separation between the layout and the content of the web document. The templates are typically used to specify the layout of the web document; the content of the web document is obtained from a different source (e.g. XML, JSON). Using template variables

we can denote the gaps in the HTML code where the content should be substituted into (see example later on this section). A template engine combines the template and the content and renders the resulting web document. Figure 5.6.2a gives a high level illustration of how a template system works.

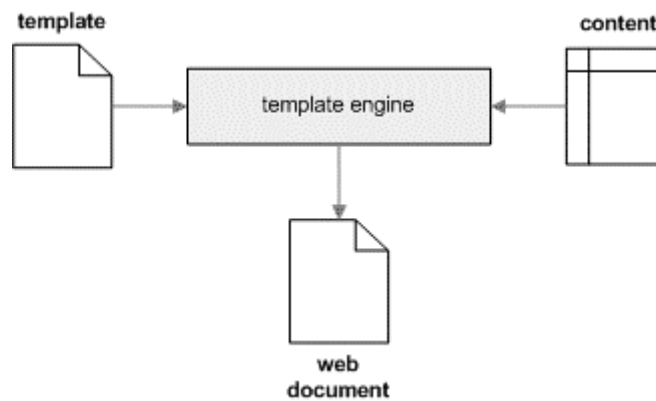
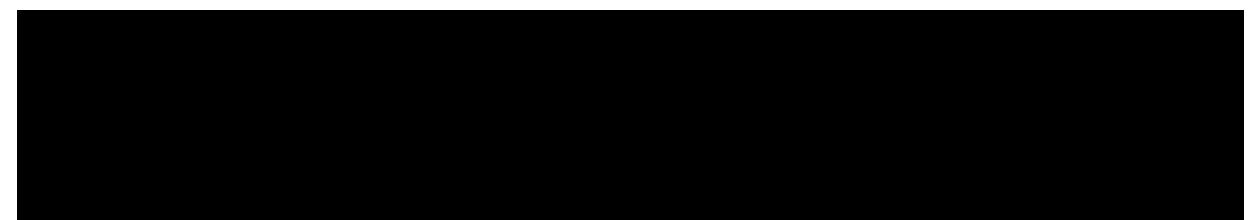
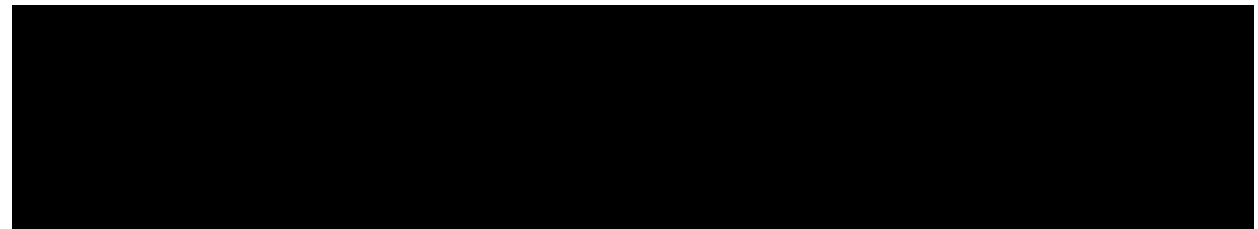
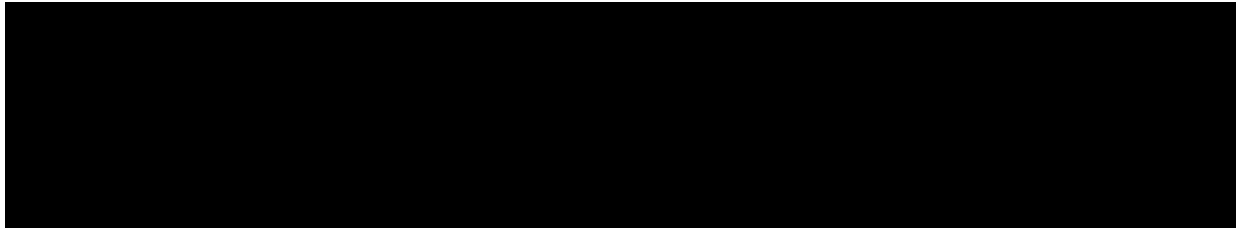
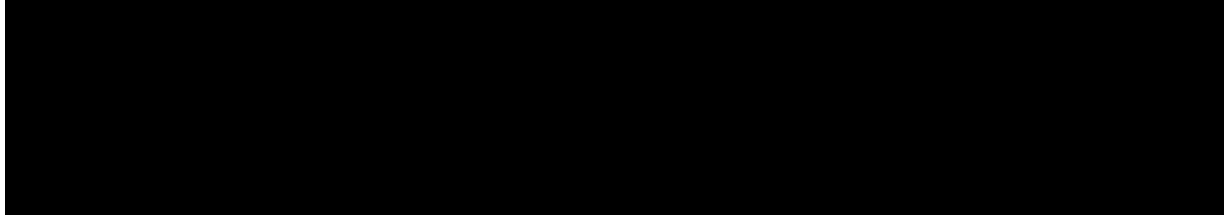


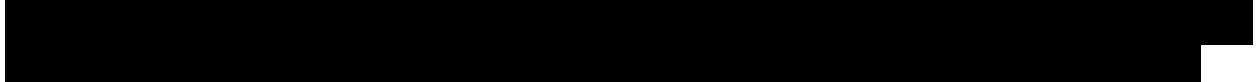
Figure 5.6.2a: Web template system





The rendering process

A key task of the client application is to render the user interface of GCL Viewer. We recall that the user interface of GCL Viewer is constructed as a web document. The presentation of the user interface is defined with HTML and the behavior (interaction) with JavaScript. The client application implements a renderer that generates the user interface. The renderer takes the enriched data (see section 5.1.2) and [REDACTED] templates as input and generates the HTML that constitutes the user interface. In figure 5.6.2b we present an illustration depicting the rendering process. The first step of the process entails the generation of the enriched data, which is taken care of by the backend component. The enriched data is then converted from the protocol buffer format to the JSON format by the frontend component. [REDACTED]



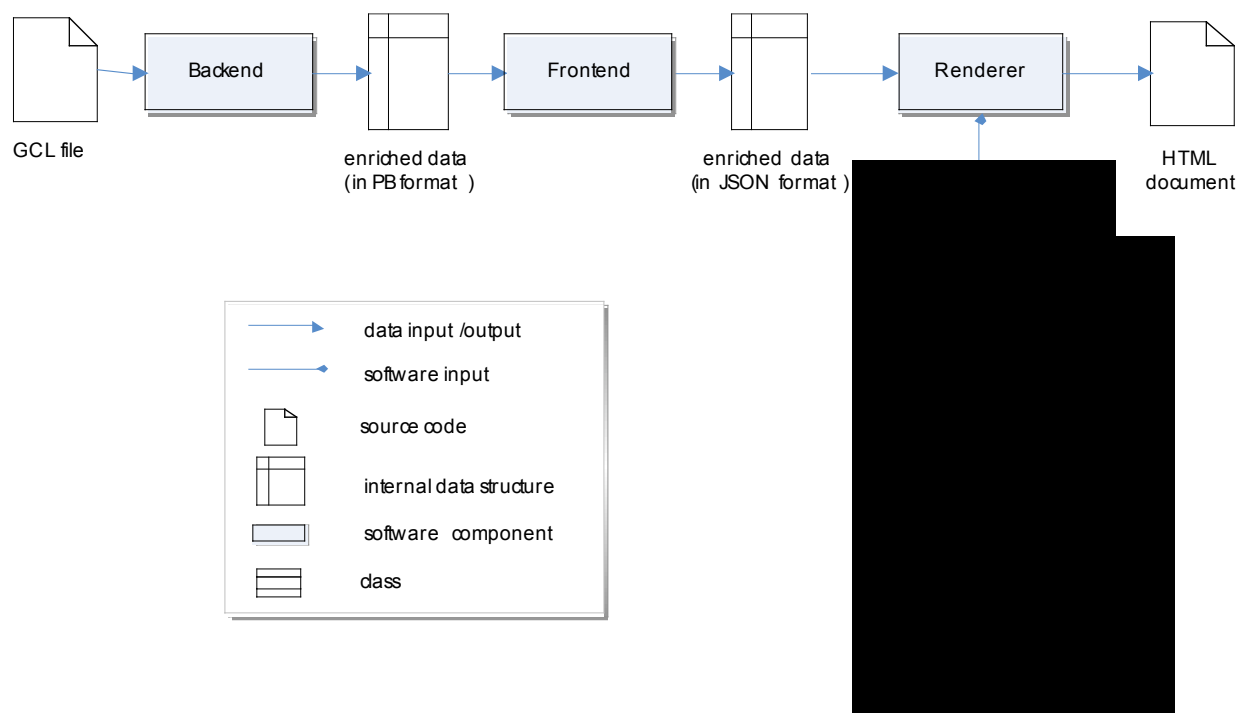


Figure 5.6.2b: Rendering process

The user interface is constructed out of a set of building blocks, which we denote as user interface elements (UI elements). Examples of UI elements are a tuple, a popup layer, and a color picker. The presentation of a UI element is specified in HTML. The renderer generates the UI elements one by one and builds up the user interface incrementally. To generate a UI element, the renderer needs two inputs: a visual specification describing how to render the UI element and the data that is going to be visualized by the UI element (The latter is optional as some UI elements are self contained entities that do not visualize data, e.g. a color picker). The visual specification of a UI element is defined in a template file using HTML, while the data to be visualized is contained in the enriched data that was retrieved from the frontend server. In figure 5.6.2c we show how a UI element is generated from the specification defined in the template and data contained in the enriched data. In this example we show a simplified version of a UI element that is used to visualize a tuple. The visual aspects of the UI element are defined in a template file using HTML. In a template we can define template variables (denoted with a dollar sign) that can be substituted with external data. In the example template of figure 5.6.2c we see two template variables, `$attribute` and `$parent`, which are later substituted by corresponding values from the enriched data. The result is an HTML fragment that specifies the UI element. In the lower part of the figure we show how each line of code of HTML contributes to the UI element.

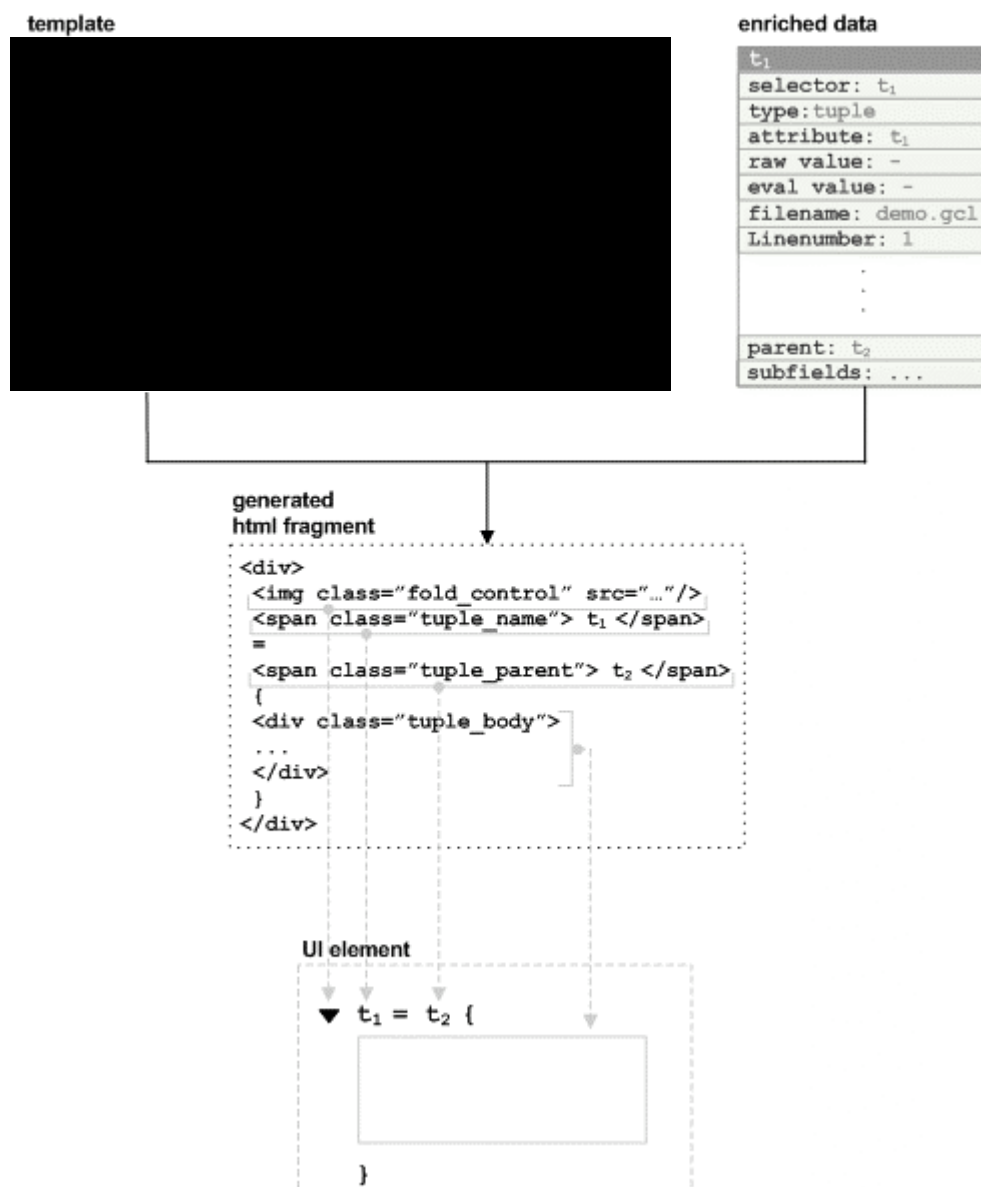


Figure 5.6.2c: An example of rendering a UI element

Finally, we add one more level of separation of code. Although we could specify the visual properties (e.g. color, font, and layout) of a UI element in HTML using HTML attributes, we choose to define these properties outside the HTML code using Cascading Style Sheets [CSS]. This makes it easier to make (quick) adaptations to the user interface (e.g. change a color of an element), and keeps the HTML code in the template compact. The visual properties defined in the CSS file are linked to the HTML code using the `class` attribute. In figure 5.6.2c we see how we assign the CSS class names *fold_control*, *tuple_name*, *tuple_parent*, and *tuple_body* to HTML elements.

5.6.3 Detailed design

In figure 5.6.3 we present a class diagram depicting the main classes of the client application.

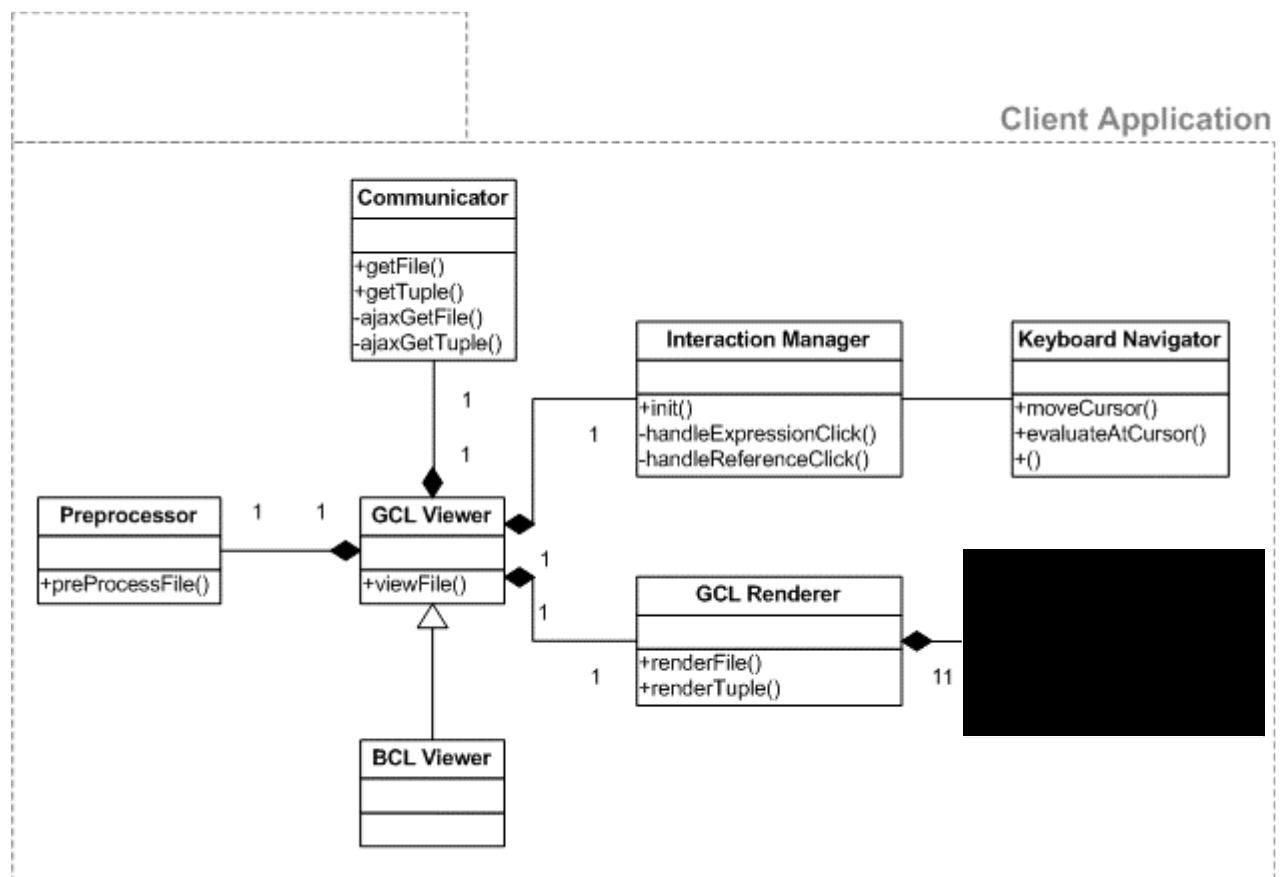


Figure 5.6.3: UML diagram for the client application

GCL Viewer

This is the main class of the client application. It is designed according to the Mediator design pattern [Gamma95] with the goal to promote independent classes in the client application. The classes *Preprocessor*, *Communicator*, *Event Handler* and *GCL Renderer*, communicate via this class rather than directly. To that purpose, the *GCL Viewer* class owns instances of the other classes and manages the communications between them. The Mediator pattern leads to reduced dependencies and lower coupling between the classes.

Preprocessor

This class is responsible for preprocessing the enriched data before the rendering phase starts. The preprocessing consists of adaptations to the enriched data that are necessary to generate the visualization described in chapter 4. Examples of these adaptations are truncating long expressions and rewriting imports to use relative paths. We note that this work could have also been carried out in the backend when the enriched data was constructed. However, this would violate our goal to separate presentation from logic code. Moreover, these adaptations are specific to the implementation of our client application. Other implementations of the client application should be free to adapt the enriched data to their own needs.

GCL Renderer

The task of the renderer is to generate the visualization that is specified in chapter 4. It constructs an HTML representation of a GCL configuration given its enriched data. [Redacted]

Communicator

This class provides the communications with the frontend server. It uses the XMLHttpRequest object that is provided by the browser to offer asynchronous communications with the frontend server. The communications are used to fetch the enriched data from the frontend server, which is needed to render the visualization.

Interaction Manager

This class implements the interactive behavior of GCL Viewer. It uses event listeners to detect user actions (e.g. mouse clicks). These event listeners are attached to the UI elements that need to provide interaction (e.g. a UI element visualizing an expression needs to react to a mouse click). The behavior of the interaction is specified in an event handler, which is called and executed upon the fire of the event.

Keyboard Navigator

This class implements the keyboard navigation of GCL Viewer.

5.7 Summary

We have designed GCL Viewer as a Rich Internet Application, which allows us to benefit from the advantages of web applications (e.g. low deployment costs, portability), while maintaining a user experience similar to desktop application (a rich and responsive user interface). Our system applies the client-side rendering technique and uses a JavaScript templating system for rendering. The client-side rendering and the object oriented JavaScript application proved to be important design decision that improved the modifiability of our application, allowing us to add new functionality more easily.

The client-server architecture allows us to improve the performance of the system by outsourcing some of the processing tasks to servers. Throughout the system we maintain a separation between presentation and business, decreasing the dependencies between the components, and promoting the modifiability of the system. And finally, we have designed the architecture of the system to facilitate the reuse of our tool and its components.

6. Evaluation

In this chapter we present the results of the survey, which we have conducted to evaluate the solution proposed in this thesis. In section 6.1 we describe the evaluation goals of our survey, in section 6.2 we describe the set up of the survey, and in section 6.3 we present and analyze the results of the survey.

6.1 Evaluation goals

The aim of the survey is to evaluate the success of our solution. Specifically, we aim to measure to what extent GCL Viewer helps the users in their work with GCL configurations, and secondly, we aim to find out which features of the tool are perceived as being useful by the users in relation to their work with GCL configurations.

6.2 Survey design

The survey consists of three sections. In the first section we evaluate to what extent GCL Viewer helps the users in their work with GCL configurations and in the second section we evaluate the usefulness of the features of GCL Viewer. We use the last section to record some demographics about the survey participant.

Section 1

In the first section of the survey we asked the users to state to what extent they agree or disagree with the following statements:

1. *GCL Viewer gives me a better understanding of the GCL language*
2. *GCL Viewer gives me a better understanding of GCL configurations*
3. *GCL Viewer helped/helps me to learn GCL*
4. *GCL Viewer helps me find bugs in configurations*
5. *GCL Viewer helps me to validate configurations*
6. *GCL Viewer simplifies the maintenance of configurations*
7. *GCL Viewer helps me to try out new features in GCL*
8. *GCL Viewer saves me time in working with configurations*

The answers were specified using a Likert Scale from 1 to 5, where 1 indicates strongly disagree and 5 strongly agree.

Section 2a

In this part of the survey we asked the users to rate the usefulness of the features of GCL Viewer on a scale from 1 to 5, where 1 indicates not useful and 5 very useful. We accompanied each feature with a brief explanation, in order to make sure that the users understood which feature was meant. Because it is possible that some users have not used all the features, we also provide the option to the user to state this.

Section 2b

In this part of the survey we asked the users to select (at most) 4 features of GCL Viewer that they use most in their work with GCL configurations.

Section 3

The survey concludes with some demographic questions asking the users to specify their function within the company, their experience with configuration languages, their experience with GCL Viewer, and their experience with configuration tools in general. We also provided the users an open field for general remarks and feedback on the tool.

6.3 Set up

Our aim was to obtain a random selection from the population of GCL Viewer users. We provided the survey via the Web, and put a link to it on the initial page of GCL Viewer, which is shown when GCL Viewer is started. We have also sent an invitation email to the mailing list for GCL users. We have kept the survey available for 2 weeks; after that we shut down the survey and removed the link to it from GCL Viewer

6.4 Results and analysis

We had 21 participants that filled out the survey. The results of the survey are presented in this section. The analysis of the results is split into three parts. First we present and analyze the results for all the participants (section 6.4.1). We continue with an analysis of the results where we have grouped the participants on their experience with GCL Viewer (6.4.2). In the last part we provide an analysis of the results where we have grouped the participants on their experience with configuration languages (section 6.4.3).

6.4.1 Results for all participants

In this section we present and analyze the results of the survey for all survey participants.

Section 1

In figure 6.4.1a we present a histogram showing the average score for each statement from section 1 of the survey.

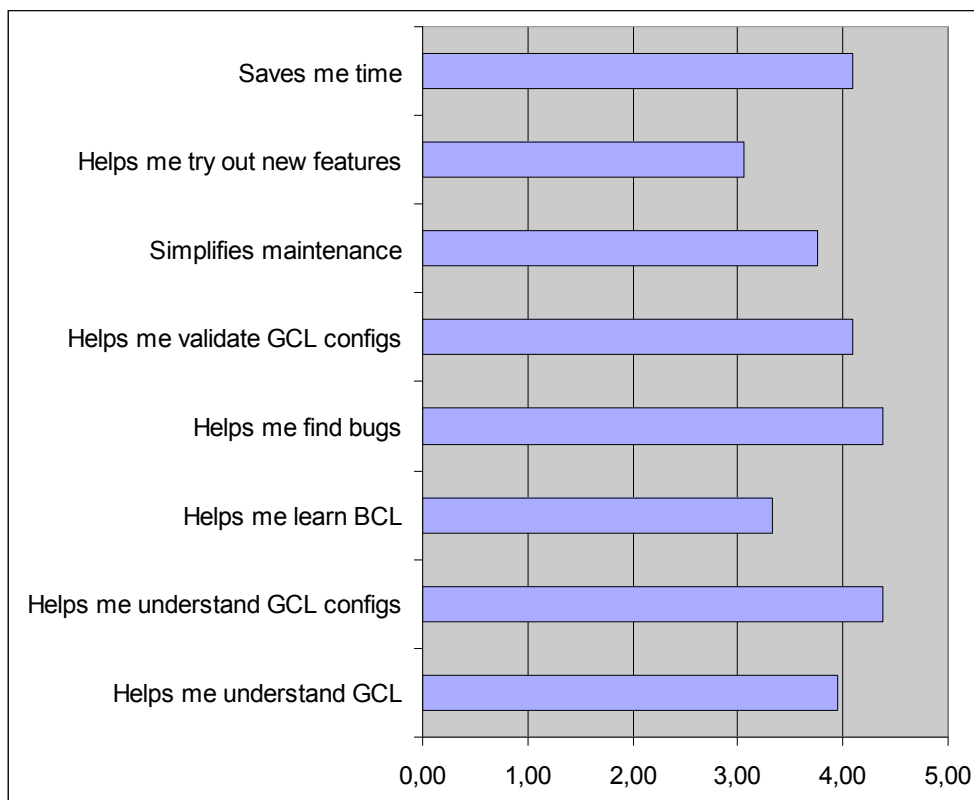


Figure 6.4.1a: Histogram showing the average scores for the statements from section 1 of the survey

These results are very good in general; all statements were rated with an average score above 3.0. The scores of two statements stand out: *GCL Viewer helps me understand GCL configurations*, and *GCL Viewer helps me find bugs in GCL configurations*. Especially the first one is a great result as it reflects one of the main goals of this thesis. The statement that received the lowest rating is *GCL Viewer helps me try out new features*. A possible explanation for this is that at the time of the survey the development of GCL Viewer had been paused already for 5 months, which means that new features of GCL were not being reflected in GCL Viewer. Also the statement *GCL Viewer helps me learn GCL* received relatively a low score. This can be explained by the fact that there is good documentation around for GCL in the form of manuals and tutorials. It is a common practice within the company to learn new material via these tutorials; new employees are expected to go through these tutorials during their first weeks at the company. A way to improve this is to integrate the viewer into these resources (e.g. use it to provide an enriched view on the examples in the tutorials), instead of presenting the viewer as a separate learning tool.

Overall, we can conclude that the tool succeeds well in helping users in their work with GCL configurations.

Section 2a

In figure 6.4.1b we present a histogram showing the average scores that were given to the individual features of GCL Viewer.

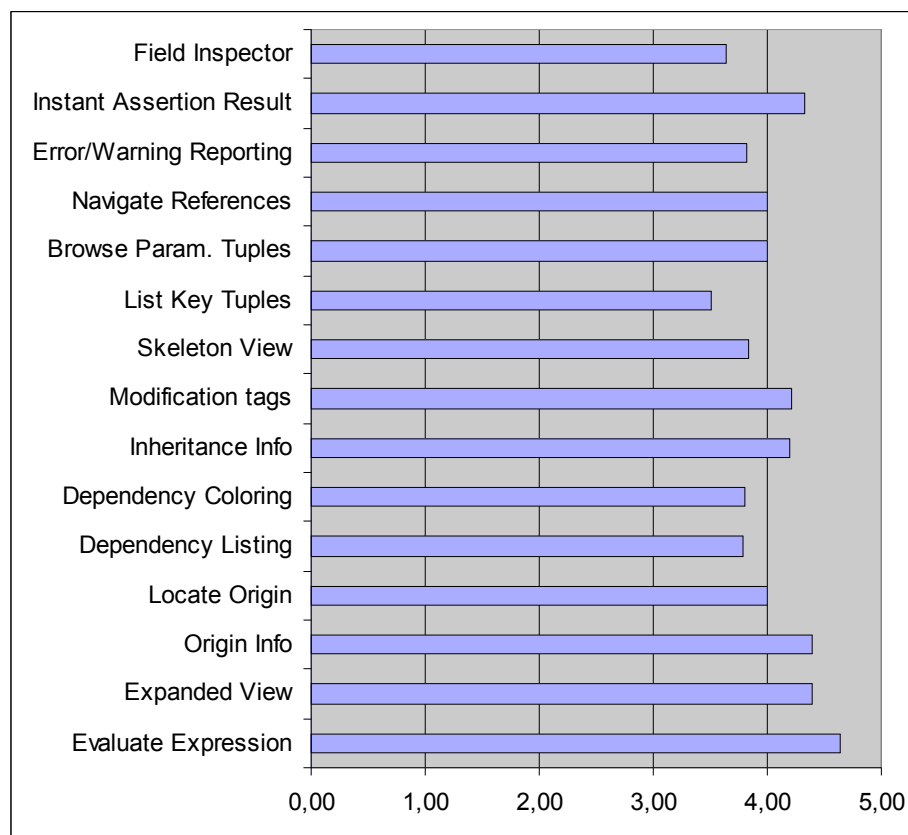


Figure 6.4.1b: Histogram showing the average scores that were given to the features of GCL Viewer

We observe that all the features have an average score of at least 3.5. This is a very satisfactory result, indicating that the solutions that we designed and described in chapter 4 are generally being considered as useful by the users. We also observe that none of the features stands out with a negative score. Further, we can observe that six features received an average score above average (4.0): *Evaluate Expression*, *Expanded View*, *Origin Info*, *Inheritance Info*, *Modification Tags*, and *Instant Assertion Result*. It is interesting that three of these features (*Evaluate Expression*, *Expanded View* and *Instant Assertion Result*) were designed to provide the user a view on what a configuration is actually defining. This conforms to the fact that the users had expressed during our initial meetings to be mainly interested in what a configuration is actually defining (see chapter 3). Further, we note that the *Origin Info* feature is strongly related to the *Expanded View* feature: the *Expanded View* shows all the fields of a tuple including the inherited and important ones (resulting in a group of fields with a mixture of origins); in such a view the *Origin Info* is useful to find out where a certain field comes from. The other two features that stand out, *Inheritance Info* and *Modification Tags*, are features designed to get insight into inheritance issues. This suggests that inheritance issues are a major contributor to the misunderstanding of GCL configurations.

In figure 6.4.1c we present a histogram depicting for each feature how many participants never to have used the feature.

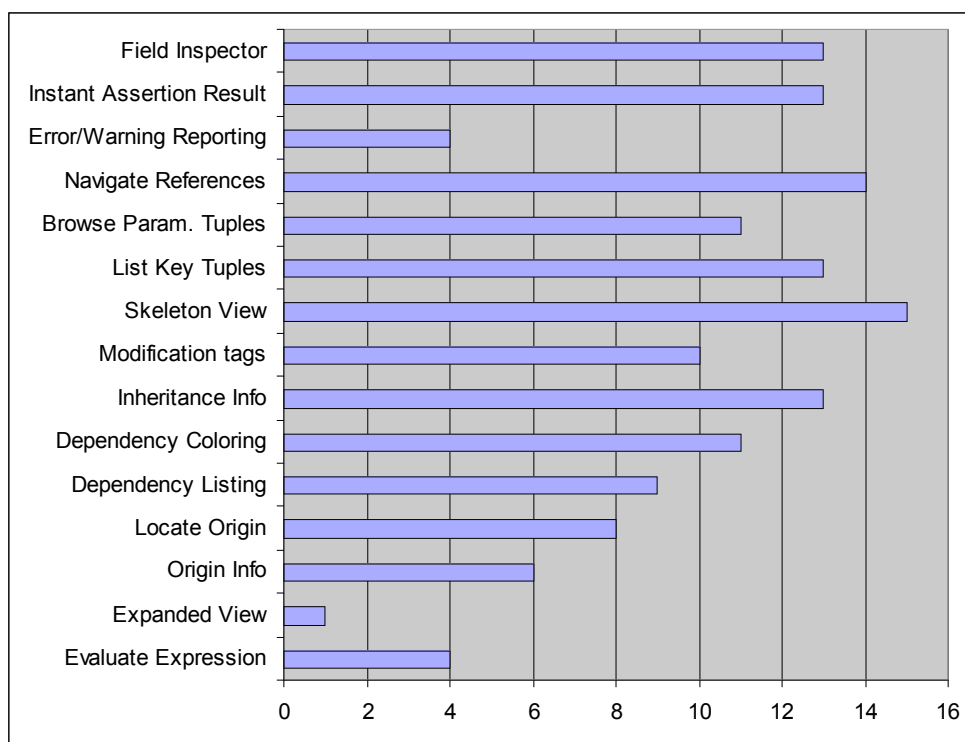


Figure 6.4.1c: Histogram showing how many participants indicated never to have used the feature

We see here a wider score distribution among the features compared to the histogram of figure 6.4.1b. We observe that the features *Evaluate Expressions*, *Expanded View*, and *Origin Info*, which had shown an above average score in figure 6.4.1b, have been rated by at least 15 participants. Compared to the other three features that received an above average score, *Inheritance Info* (8 participants), *Modification Tags* (11 participants), and *Instant Assertion Result* (8 participants), we can say now with more certainty that these features are successful.

Another observation is that many features have been stated to never have been used, of which six features never have never used by more than half of the participants. More research has to be

conducted to find out why most of the features are not used by all users. A possible explanation is that most of the features are unknown to the user. For 4 of these features (*Field Inspector*, *List Key Tuples*, *Skeleton View*, and *Inheritance Info*) this could be explained by the fact that they are not shown by default in the viewer, but need to be accessed via the menu or keyboard. This observation is emphasized by the fact that *Error/Warning Reporting*, which uses visual hints to indicate its presence, is known to the majority of the users (17 participants). This means that we should improve the communication about the features to the users of the tool and integrate visual hints into the tool to indicate the availability of these features. Another observation is that two features that are visible to the user by default (*Instant Assertion Result* and *Navigate References*), are also not so commonly used. A possible explanation is that they comprise language constructs that are not so commonly used in configurations as expressions and inheritance. Another possibility is that the issues these features has but more research has to be conducted to find this out.

Section 2b

In figure 6.4.1d we present a histogram that shows how often a feature was selected by a participant to be used most in working with GCL configurations.

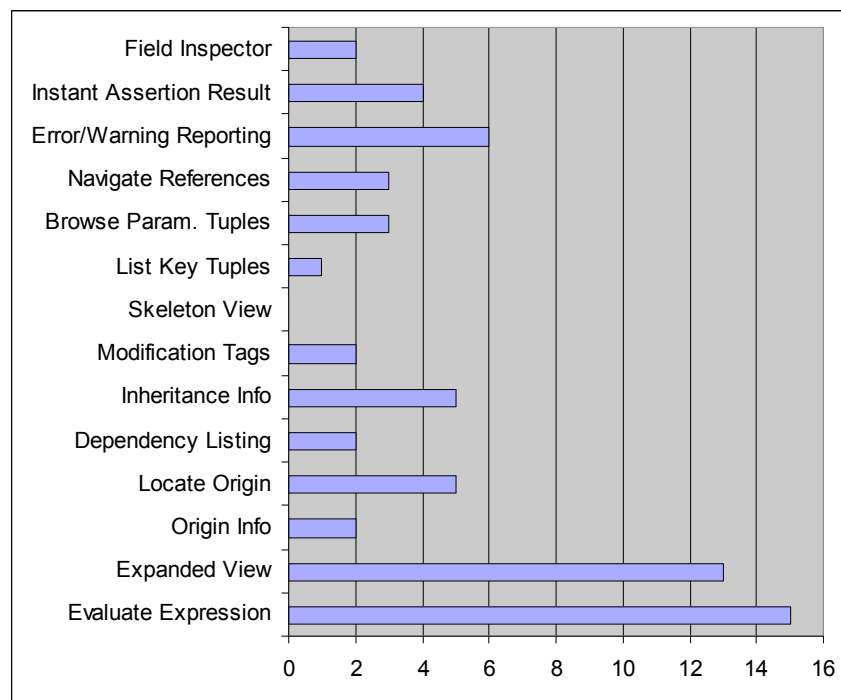


Figure 6.4.1d: Histogram showing how often a feature was selected by a participant to be used most in working with GCL configurations

The results from this question emphasize our findings from figure 6.4.1b; the features that received the highest average rating over there, also have the highest usage frequency. Further, we can observe here that the features *Expanded View* and *Evaluate Expression* stand out much stronger from all the other features. Therefore, we can conclude that these two features are the most useful ones in GCL Viewer.

Summary

The participants regard the tool as being very useful in their work with GCL configurations. The tool is especially valued for helping understand GCL configurations and finding bugs in GCL configurations. Another satisfying result is that users have stated that the tool helps them save time in working with GCL configurations.

All the features received an above average rating, but *Expanded View* and *Evaluate Expression* clearly stand out with an above average score throughout all the questions. This is an interesting result, because they form two of the three features that were designed to provide an explicit view, i.e. to show what a configuration is actually defining (see section 4.1). The fact that the third feature of this kind, *Browse Parameterize Tuples*, received a relatively lower score (4.0) could be explained by the fact that it addresses a language construct that is less commonly applied than the other two. This speculation is confirmed by the histogram in figure 6.4.1c, where we can see that more users have denoted the latter as a feature they have never used.

The survey results also indicate that the features that have been addressed to get insight into inheritance issues are valued more by the users.

6.4.2 Results grouped by experience with GCL Viewer

In this section we present the results of the survey grouped by the experience of the participant with GCL Viewer. At the time of the survey, GCL Viewer had been available for about 8 months. We have asked the participants to state how long they have experience with GCL Viewer (less than 3 months, 3 to 6 months, or longer than 6 months). We would like to compare the results of novice users and experienced users. Therefore, we study the results of two groups; the users that have less than 3 months of experience with GCL Viewer, and users that have longer than 6 months of experience. We have 7 participants in the first group and 8 participants in the second. The results of the survey are presented in the same order as in the previous section.

Section 1

In figure 6.4.2a we present a histogram showing the average score for each statement from section 1 of the survey.

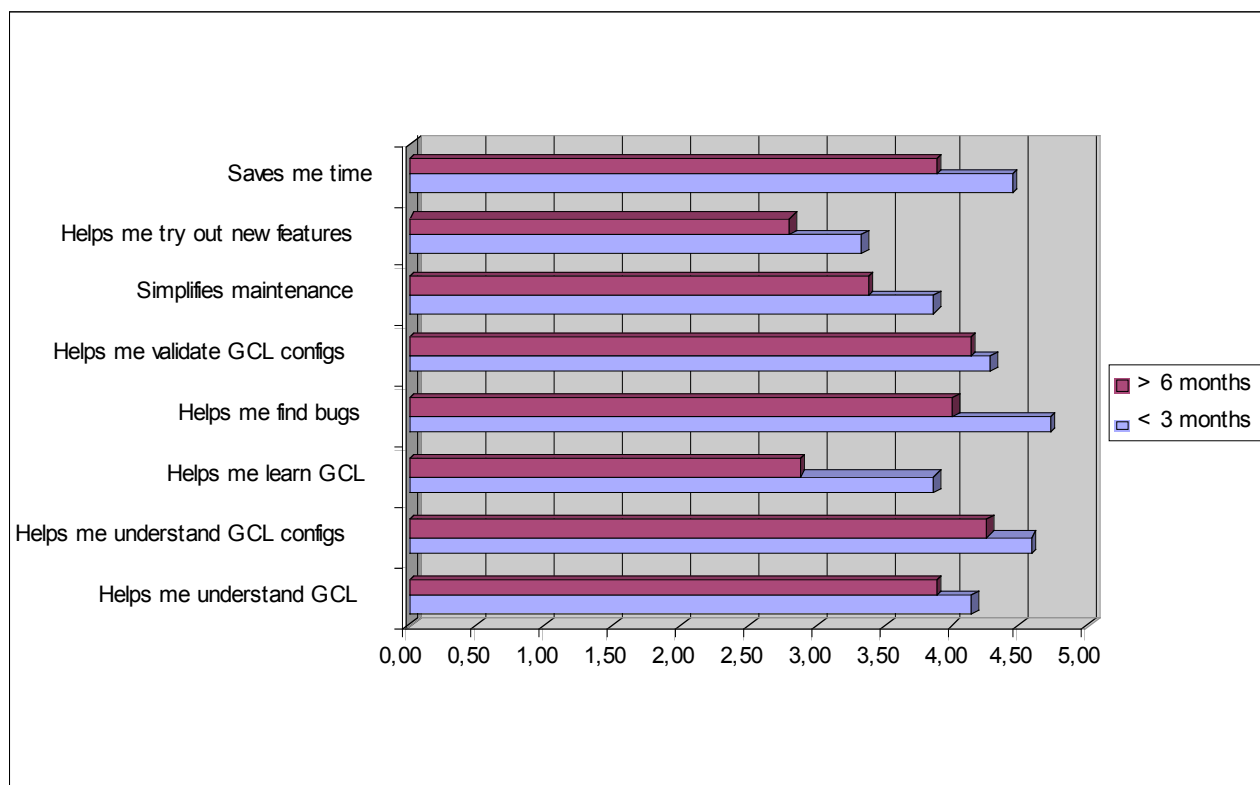


Figure 6.4.2a: Histogram showing the average scores for the statements from section 1 of the survey, grouped by experience with GCL Viewer

We first observe that the participants that are relatively new to GCL Viewer give on average a higher score on all statements. A possible explanation is that users in general are more enthusiastic when having discovered something new. The largest differences in score are observed for *GCL Viewer helps me learn GCL configurations* (1.0) and *GCL Viewer helps me find bugs*. A possible explanation for the latter is that users make fewer mistakes in their configuration as they gain more experience with GCL (Viewer). Another observation is that beginners agree more on the statement *GCL Viewer saves me time in working with configurations* in working. Further, we see that two statements score relatively low with the more experienced participants: *GCL Viewer helps me try out new features* (2.8) and *GCL Viewer helps me learn GCL configurations* (2.9). This can be explained by the fact that a more experienced user is already familiar with GCL, and is therefore less concerned with learning GCL compared to a beginner.

Section 2a

In figure 6.4.2b we present a histogram showing the average scores that were given to the individual features of GCL Viewer, grouped by experience with the tool.

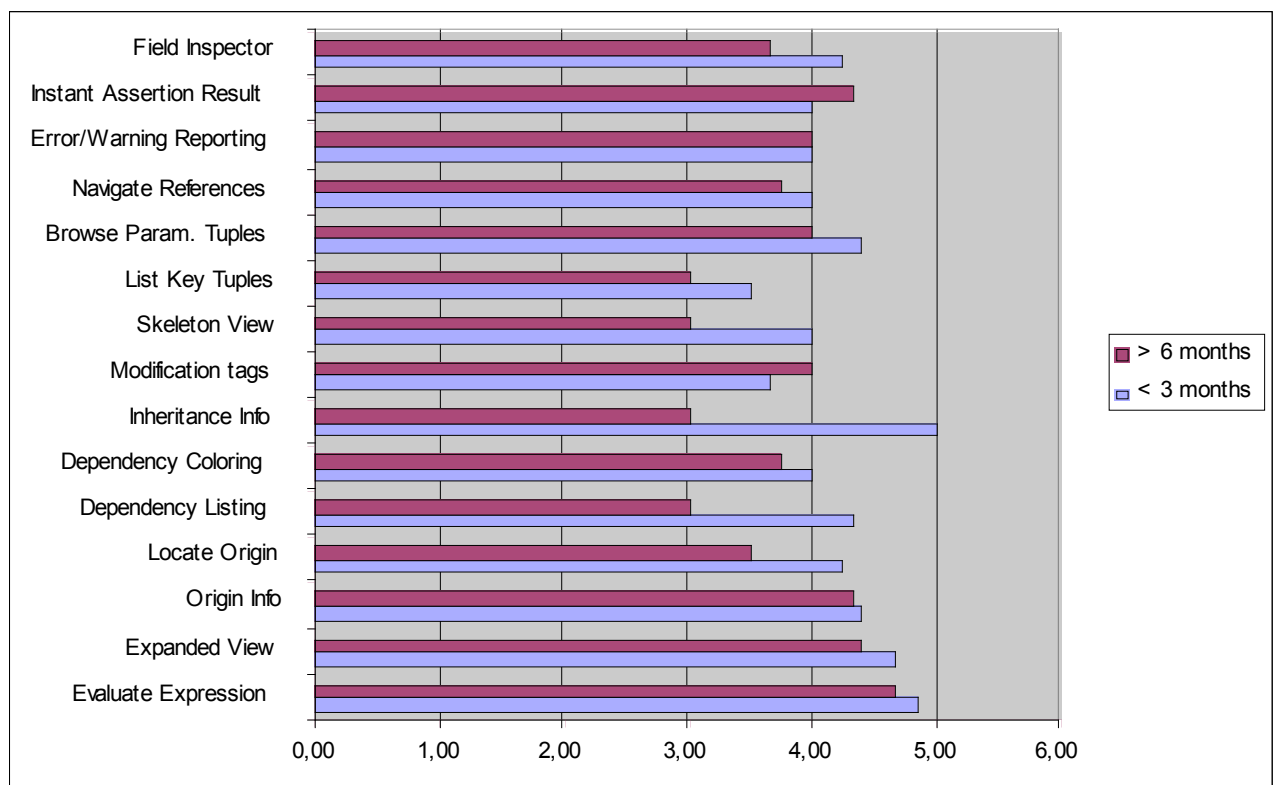


Figure 6.4.2b: Histogram showing the average scores that were given to the features of GCL Viewer, grouped by experience with the tool

Also here we observe that participants that are relatively new to the tool give higher scores compared to the more experienced users. An important observation is that the features that have been identified in the previous section to be the most successful (*Evaluate Expression*, *Expanded View*, and *Origin Info*) score very high with both groups of users (≥ 4.4). We cannot draw significant conclusions for the largest difference in score (the *Inheritance Info* feature), because only 2 participant in the first group and 3 in the second group have rated this feature. The same argument holds for the *Skeleton View* feature. More research has to be conducted to explain the relatively large difference (1.3) for the *Dependency Listing*.

Section 2b

In figure 6.4.2dc we present a histogram that shows how often a feature was selected by a participant to be used most in working with GCL configurations. The results are grouped by experience with GCL Viewer.

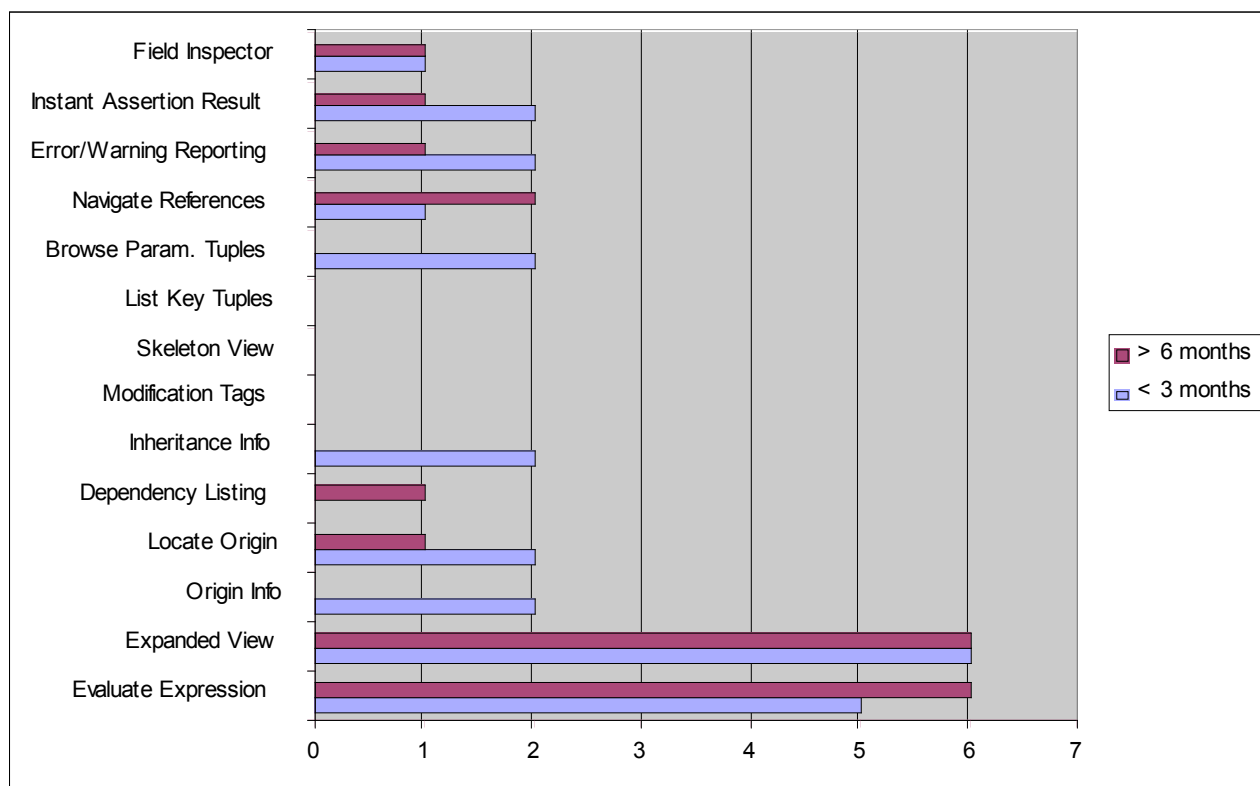


Figure 6.4.2c: Histogram showing how often a feature was selected by a participant to be used most in working with GCL configurations, grouped by experience with GCL Viewer

Also here we observe that both *Expanded View* and *Evaluate Expression* score very well in both groups compared to the other features. The differences in score for the other features are small.

6.4.3 Results grouped by experience with configuration languages

In this section we present the results of the survey grouped by experience with configuration languages. We had asked the participants to state how long they have been working with configuration languages (less than 1 year, 1 to 3 years, or longer than 3 years). We would like to compare the results of the participants that have less than 1 year of experience with configuration languages with the group of that has longer than 3 year of experience. However, we only had 2 participants in the > 3 years group, hence we decided to compare the < 1 year group with the ≥ 1 year group. We have 13 participants in the first group and 8 participants in the second. The results of the survey are presented in the same order as in the previous section.

Section 1

In figure 6.4.3a we present a histogram showing the average score for each statement from section 1 of the survey.

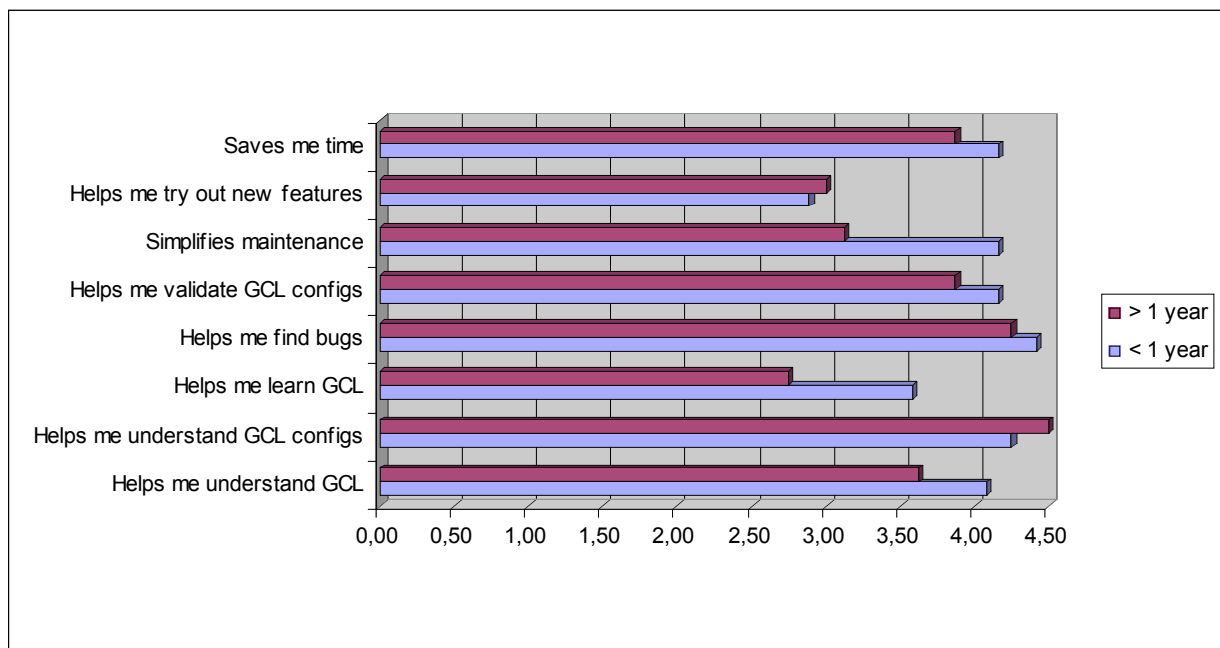


Figure 6.4.3a: Histogram showing the average scores for the statements from section 1 of the survey, grouped by experience with configuration languages

For the majority of the statements there is no significant difference in score. The main observation is that the < 1 year group finds the tool to be more useful in helping them learn and understand GCL. They also find the tool more helpful for maintenance tasks compared , in which understanding of GCL (configurations) plays an important role.

Section 2

In figure 6.4.3b we present a histogram showing the average scores that were given to the individual features of GCL Viewer, grouped by configuration languages experience.

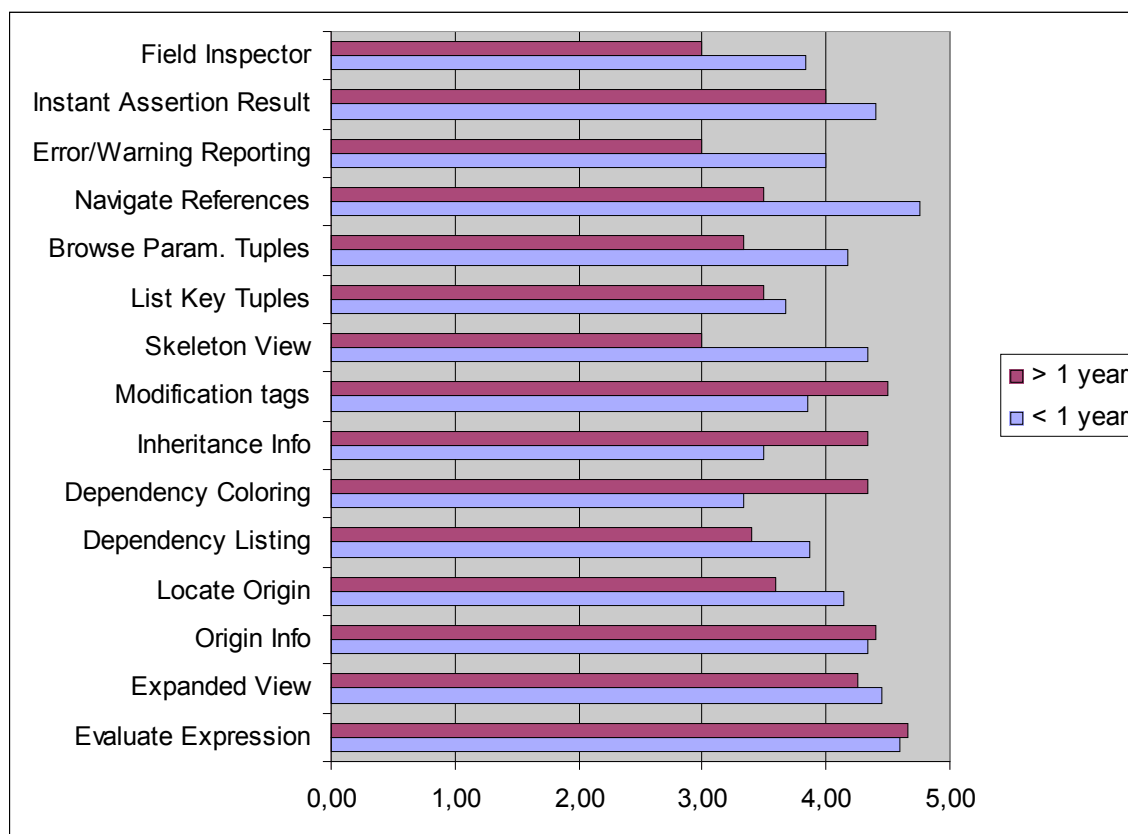


Figure 6.4.3b: Histogram showing the average scores that were given to the features of GCL Viewer, grouped by experience with the tool

Also here we observe that *Evaluate Expression* and *Expanded View* receive above average scores by both groups. Same holds for *Origin Info* which is a feature strongly related to *Expanded View*.

We observe that for most of the features the < 1 year group gives higher ratings to the features than the > 1 year group. This indicates that users that are relatively new to configuration languages find the tool more useful.

The features *Inheritance Info*, *Modification Tags* and *Dependency Coloring* score higher with the >1 year group. A possible explanation is that these features address more advanced issues (e.g. inheritance), which is applied more intensively by users with more experience with configuration languages.

7. Related work

In this chapter we discuss work related to GCL Viewer. We start with a brief history on visualizing computer programs (section 7.1) and introduce the terminology that is used in literature (section 7.2). In section 7.3 we use a software visualization taxonomy to classify our tool. In section 7.4 we provide an overview of tools that share characteristics with GCL Viewer. In section 7.5 we give a brief overview of other interesting software visualization tools. We conclude with section 7.6 in which we describe some work that relates to our approach to integrate our tool into web services.

7.1 History

The importance of visual representations in understanding computer programs is a concept that has been known for a long time. Goldstein and von Neumann [Goldstein47] demonstrated the usefulness of flowcharts, while Haibt [Haibt59] developed a system that generates these flowcharts automatically. The introduction of the bitmapped display and window interface technology in the 1980's started the modern research on visualization of computer programs. Since then research in this area has led to the development of many visualization tools for computer programs of which [Brown88], [Esdt88], and [Stasko90] are some of the best examples [Price93]. A more detailed overview of the history of visualization of computer programs can be found in [Stasko98].

7.2 Terminology

The term Program Visualization (PV) was introduced in the late 1980s by [Myers86] to denote the field of study that is concerned with using graphical techniques to display programs. We prefer to use the term Software Visualization (SV) which has been introduced by [Price92], as it is a wider definition which includes PV as a sub area. Software Visualization is defined as the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction technology to facilitate both the human understanding and effective use of computer software [Price93]. Software Visualization has two main sub areas, algorithm visualization and program visualization. Program visualization can be divided into static code visualization, static data visualization, data animation, and code animation. GCL Viewer is a static code visualization tool.

7.3 Classifying GCL Viewer

In this section we aim to put GCL Viewer in the context of related Software Visualization work by classifying it according to the Principled Software Visualization Taxonomy [Price93]. Price's taxonomy defines six categories that together describe and classify a Software Visualization tool:

- *Scope*: Describes the range of programs that the SV tool can take as input for visualization.
- *Content*: The subset of information about the software that is visualized by the SV tool.
- *Form*: The characteristics of the output of the SV tool.
- *Method*: Describes how the visualization is specified.
- *Interaction*: The way the user of the SV tool interacts with and controls it.
- *Effectiveness*: Describes how well the system communicates information to the user.

We will describe the specifics of GCL Viewer for each category of the taxonomy.

Scope

GCL Viewer takes as input GCL programs and programs of languages derived from GCL [REDACTED]. GCL Viewer does not impose a restriction on the kind of GCL program that can be visualized. GCL Viewer is designed to scale up easily to handle large examples of GCL programs.

Content

GCL Viewer visualizes GCL programs on a source code level.

Form

The visualization that is output by GCL Viewer consists of an HTML marked up document describing the source code of the GCL Program. The visualization provides an enriched view on the source code by using text formatting (e.g. syntax coloring, indentation) and widgets (e.g. tags, buttons, tooltips, drop-down lists) in the presentation. Other characteristics of the presentation are:

- The visualization presents an explicit view on the code, expanding tuple definitions and resolving expressions.
- The initial view provides shows the source code only. Details are available on demand. In most cases visual cues are used to indicate the presence of additional information (e.g. *error icons*, *modification tags*).
- The visualization of GCL Viewer uses color also to associate elements of the visualization (e.g. *dependency coloring*), to highlight an element in the visualization (e.g. *resolving references*), and to convey information (e.g. *instant assertion result*).
- GCL Viewer provides by default a fine-grained view (showing the source code entirely) but offers mechanisms (e.g. *skeleton view*) to filter out details in order to obtain a more coarse-grained view.
- GCL Viewer provides the ability to temporarily hide sections that are not of immediate interest (*foldable tuples*).

Method

GCL Viewer generates the visualization automatically. The visualization is obtained by parsing the program first, and computing the enriched data while traversing the parse tree. The enriched data comprises all the information that is needed to render the visualization.

Interaction

GCL Viewer supports interaction via both a pointing device and a keyboard. Both types of devices can be used to browse through the code and access all the functionality of the tool. The interaction provides analytical and validation means (e.g. evaluating expressions, inspecting variables). The tool uses the hypertext technology to support navigation within a program (e.g. relative references) and between programs (e.g. external references). The tool does not provide facilities to script interactions.

Effectiveness

We have conducted a user evaluation to determine the effectiveness of our tool. We refer to chapter 6 for the results of this evaluation. The tool has been in production use for seven months, and has been used since then by over 1100 users. The daily user base is 30 on average.

7.4. Related tools

In this section we describe Software Visualization tools that share characteristics with GCL Viewer.

7.4.1 Pretty printing

In the 1970's Ledgard introduced the concept of pretty-printing [Ledgard77] to describe the use of spacing, indentation, and layout to make source code easier to read in a structured language. Another approach was literate programming introduced in the WEB system [Knuth84], which combines documentation and program in one document using a markup language. The SEE Program Visualizer [Baecker90] takes a set of C programs and automatically formats a program book out of them. GCL Viewer uses also pretty printing techniques to improve the readability of the source code (e.g. indentation, fixed width font). However, in contrast to GCL Viewer, these examples provide no means of interaction, and also hardly benefit from the use of color techniques to improve the understanding of code.

7.4.2 SID: A graphical browser

SID is an interactive software visualization tool for C++ programs [Craw91]. It uses knowledge based software to assist in determining which components of the program should be visible and what attributes should be used to display these components. The user also can adjust the viewing controls. SID is similar to GCL viewer in the sense that it is designed to enable the programmer to get a sense of the system as whole. It offers a control panel allows the user to adjust the view by specifying items to remove from consideration (filter), items which should stand out (enhance), and areas of interest.

7.4.3 Enhancing Code for Readability and Comprehension Using SGML

Cowan et al have explored how extra semantics information about a program (obtained during compilation) can be of assistance when reading and understanding source code [Cowan94]. Their method is to use mark up languages such as SGML to embed information about the syntax and semantics of a program in the program code, and then show these in the presentation of the program. They also use typesetting techniques to convey information about the structure of the program. In the future they want to improve navigation of source code by using hypertext links that take you for example straight to a function's definition. They have written converters to convert code of a program language to an equivalent SGML representation (e.g. a C to SGML converter). This work is very similar to GCL Viewer in the sense that both use a markup language to display source code. Another shared characteristic is that both tools obtain extra semantics information obtained during compilation to provided an enriched view on the source code. A main difference is that GCL Viewer provides an interactive visualization.

7.4.4 CodeSurfer

An example of a rich and interactive software visualization tool is CodeSurfer [Anderson01]. CodeSurfer is a static analysis tool designed to support program understanding based on the dependence-graph representation of a program [Anderson01]. CodeSurfer has features similar to GCL Viewer. The *Property Sheet* of a variable is similar to our *Field Inspector*. It provides detailed information about a variable such as where the variable was defined and where its value is used (see figure 7.4.4). The *Predecessor/Successor* feature that is designed to find answers on questions such as "How could variable *x* have gotten its value here?" is similar to the *Inheritance Stack* feature in GCL Viewer. It also provides a view showing the call graph for the program. CodeSurfer also provides features to visualize call graphs programs, which brings useful insight into procedural (imperative) languages.

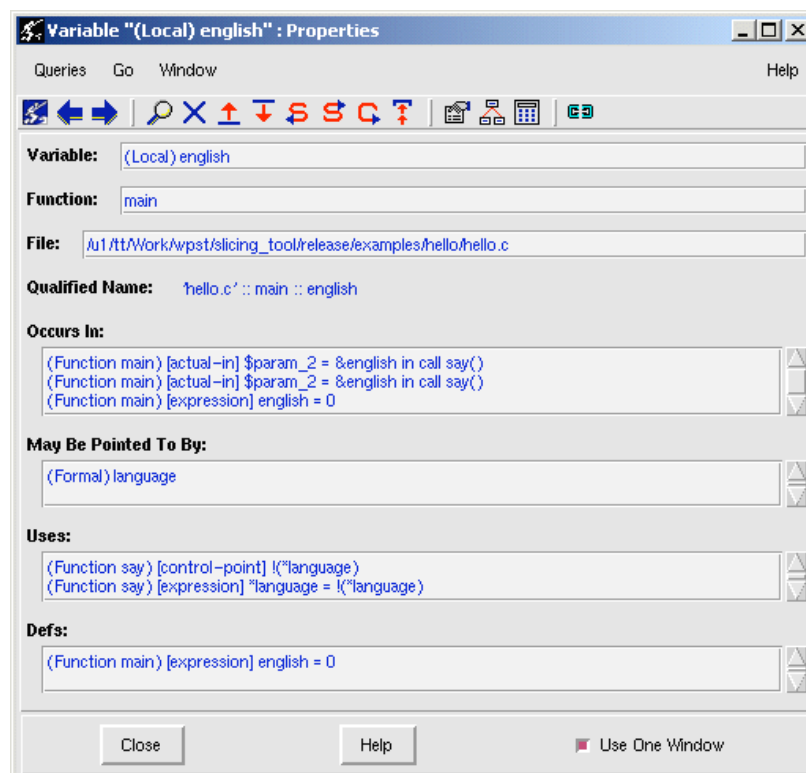


Figure 7.4.4: The property sheet feature of CodeSurfer

7.4.5 Fisheye View

The fisheye view aims to help a programmer in navigation and understanding programs by displaying those parts of the source code that have the highest degree of interest given the current focus. Jakobsen et al have conducted an empirical evaluation on the fisheye technique and concluded that overall, the participants performed software tasks faster with the fish eye view, and that they generally preferred the interface with the fish eye view [Jackobsen]. GCL Viewer offers a manual fish eye view feature (*folding tuples*), which can be used

7.5 Other software visualization tools

In this section we describe Software Visualization tools that apply a visualization different from static code visualization. Although these tools are not directly related to GCL Viewer we do list them here to provide insight in other interesting forms of Software Visualization.

7.5.1 Graph based software visualization tools

[Rigi], [SHriMP], [CodeCrawler] are tools mainly targeted at object-oriented software and visualize the high-level entities of a program such as classes, modules and packages. They provide several (graph based) views that visualize the relationship between the program entities (e.g. class relationship, method calls). These views are usually extracted by reverse engineering the source code. Another type of graph based software visualization tools generate control and data flow diagrams based on the source code.

7.5.2 Visualization of software evolution

A good understanding of software on source code level is a necessity in software development activities such as software maintenance. Insight in the evolution of software, describing the changes to code over time, is another aspect that plays an important role in software understanding and maintenance. An example of a tool that visualizes the evolution of software systems is [CVSscan05]. This tool presents a multiview environment consisting of a line-oriented display of the changing code, a view on the source code itself, and panels showing

various metrics. Such a tool applied to the GCL code base would provide additional insight into GCL programs and improve the understanding and maintenance of GCL programs further. An interesting scenario is to integrate GCL Viewer into CVSScan to provide a richer view on the source code level.

7.6 Other related work

In this section we describe related work to our approach to integrate a (software) visualization tool with web documents to provide a rich and interactive insight on the provided material.

7.6.1 Embedding visualization into electronic documents and reports.

[Jern] explains the advantages of embedding and distributing data visualization components in electronic documents and reports (especially in the scientific and engineering domain). The reader of the document can then take advantage of visualization techniques to help understand the information (obtained from experiments and simulations) provided in the document. This allows the information recipient to interactively examine the data in the same way as the original analyst. The author sees an opportunity in the increasing use of electronic documents distributed by intranet and internet. This work is related to GCL Viewer in the sense that we have designed our tool as a visualization component that is embeddable into web documents (e.g. wikis, manuals, tutorials), with the aim to provide the reader a richer view on the information. Since the output of our visualization is in HTML, it can be also integrated in emails, for example, which can help engineering teams to communicate and discuss their GCL programs.

7.6.2 Using the Web as a platform for visualization tools

The web is now being used for many more applications than just reading text documents and the browser is becoming the default user interface. [Tilley97] and [Domin97] advocate the use of the web as a platform for enabling visualization tools. A possible domain application is online education in which visualization tools (e.g. algorithm animations) are used to educate programming languages and algorithms [Roberts06, Pareja03].

8. Conclusions

In this thesis we introduced GCL Viewer as a software visualization tool for GCL. The tool visualizes GCL programs on a source code level. The tool provides an enriched view on the program and is interactive. GCL Viewer also provides a means of analyzing and validating GCL programs. Our Rich Internet Application design approach allowed us to create a tool that benefits from advantages of web applications (e.g. low deployment costs, portability, integration with web services) while achieving a user experience that resembles desktop applications closely (e.g. a rich and responsive user interface).

The primary goal of our thesis study was to improve the understanding of GCL and GCL programs. We conclude that GCL Viewer has achieved this goal successfully. We base our conclusion on three sources: the evaluation results, usage statistics of the tool, and recorded feedback from the users.

- The user evaluation results show that the tool improves the understanding of GCL and GCL programs. From the same results we also conclude that the tool succeeds in its goal to help users in their work with GCL programs: The survey participants have stated to find GCL Viewer to be helpful with maintaining, debugging, and validating GCL programs, and that it saves them time with these activities.
- Using a monitoring system that we had installed at the time of deploying GCL Viewer, we have gathered usage statistics of GCL viewer. Statistics over the last 6 months show that GCL Viewer has been used by over 1100 users, while the daily number of page views of the tool is 45 on average (see appendix B).
- GCL Viewer has been received well by its users. We have received many positive feedback from enthusiastic users via spontaneous emails and the ‘remarks’ section of the survey. An overview of these feedbacks is given in appendix C.

Below we reflect on the secondary goals of our thesis, which were formulated in the introduction of this thesis (chapter 1):

- A. The GCL Viewer project has acted as a case study on applying software visualization in an industry setting. GCL Viewer was designed to address complexity issues in a near exponential growing code base consisting of over 1.3 million lines of code, spread over 22400 files, and developed by over 2300 programmers. The highly positive results from our user evaluation indicate that a well designed software visualization tool can help significantly in understanding and managing software in such a setting.
- B. We have designed GCL Viewer as a Rich Internet Application and demonstrated that web technology has matured to allow the development of interactive and responsive software visualization tools. We have also used GCL Viewer to advocate the integration of visualization tools into web services such as online education resources (e.g. tutorials, wikis), We have set plans to integrate GCL Viewer with Google’s Codelab, which provides the online education resources to Google engineers.
- C. Based on the results of our user evaluation we can conclude that the features that were considered as being most useful are the features that provide an explicit view on the program (e.g. *expanded view*, *evaluate expressions*). It is very likely that these features play a key role in improving the understanding of GCL programs. Based on this assumption, we conclude that programmers benefit from code visualizations that provide an explicit view on implicit and hidden information. This observation can be applied to

improve the understanding of other programming languages. Most programming languages support expressions. A visualization that shows the evaluated values of these expressions can be considered as being useful to the user. Examples of other information that can be useful to be made explicit by visualizations are inherited variables and methods in a class, and global variables in a procedure definition. Finally, we conclude that the visualization techniques of GCL Viewer that were designed to provide insight into inheritance issues were generally very well rated by the users.

Summarizing we can conclude that we have succeeded in designing an online software visualization tool that achieves the goals stated in the introduction of this thesis. We believe that a key to the success of GCL Viewer is the thorough analysis of the complexity issues of GCL (chapter 3), and the early and continuous involvement of GCL users in the design process, which helped us to design a useful tool that fits their needs and addresses the main issues.

9. Future work

We need to conduct a usability evaluation of the tool. The results from this test should be used to improve the usability of the features of GCL Viewer. This includes, for example, a thorough study on which colors to use in the tool (taking color blind people in mind).

A technical evaluation of the system needs to be conducted in order to evaluate the technical design goals of the system (e.g. throughput, network bandwidth usage).

We plan to implement some additional features for GCL Viewer. One of these features is a semantical zoom feature, which can be used to vary the level of detail that is shown by the viewer. The inheritance stack feature needs to be implemented too. We also plan to develop an online application that uses GCL Viewer to provide a way to interactively experiment with GCL programming (as is illustrated in Figure 4.1a).

We would like to integrate other GCL tools into GCL Viewer in order to provide more validation functionality. This includes the GCL validator and the lint tool (checks if the code conforms to the style guide).

We would also like to integrate GCL Viewer with an existing tool that uses graphs to visualize the dependencies between GCL files. The idea is that the user can switch seamlessly between the tools. Clicking on a node in the dependency graph would open and show the corresponding GCL configuration in the viewer. And from the viewer the user can switch to the view showing the dependency graph of the configuration that is currently being viewed.

We would like to reuse GCL Viewer for other languages that are similar to GCL. For that we need to develop new backend services, but we can reuse the frontend service and client application.

10. Project Evaluation

It was a joy to work on this project. From the first day it became clear to me that I got an opportunity to work on something that is relevant to many people. The fact that my work would help many people in their daily engineering work was a great motivation. The project had also an interesting mix of algorithm design (in the backend), visualization and user experience design (in the client application), and networks and distributed system for handling the large computational load.

The project provided me the opportunity to work with fascinating people (researchers, experts, engineers), both at Google and the Eindhoven University of Technology, from whom I gained a lot of knowledge on computer science. I am very grateful for this opportunity.

The development process at the company was very result driven. It pushed for the release of early prototypes. This prototype based engineering provided useful input from our users throughout the development process, which allowed us to develop a result satisfying the users. However, this approach was in some cases conflicting with the academic approach where intensive research and design is done prior to implementation. I do believe that the more experience I have in developing software the better this can go hand in hand. Especially at the beginning of the project I had to invest significant time to cope with a steep learning curve that came with learning new systems and new programming languages.

Peer code reviewing is useful and helps in assuring software quality. Further, all the code that we wrote had corresponding unit tests. I learned how this plays a crucial role in helping prevent other programs (that depend on your code) to fail, and how it prevents that unnoticed changes in the GCL library, for example, lead to failures in your program.

A design decision that had a very positive effect was to switch our system to a client-side rendering technique and an object-oriented JavaScript approach. This improved the modifiability of the system significantly. Adding new features became easier and could be done more efficient compared to our first prototype, which applied the server-side technique.

It is satisfying to see your own creation being used enthusiastically by the people around you. It was also very motivating to receive spontaneous feedback, in person or via email, from people in offices all over the world.

References

- [Air] Adobe Air - <http://www.adobe.com/products/air/>
- [Anderson01] Software Inspection Using CodeSurfer - Paul Anderson, Tim Teitelbaum – 2001
- [Baecker90] Baecker, R. M. & Marcus, A. (1990). *Human Factors and Typography for More Readable Programs*. Reading, MA: Addison-Wesley.
- [Bass] Software Architecture in Practise (Second edition) – *Len Bass, Paul Clements, Rick Kazman*
- [Bozzon06] Conceptual Modeling and Code Generation for Rich Internet Applications - Alessandro Bozzon, Sara Comai, 2006
- [Brown] Color and Sound in algorithm design - Marc H. Brown and John Hershberger
- [Brown88] Brown M. H. - Exploring Algorithms Using Balsa II. - IEEE Computer, 21(5): 14-36.
- [CVSscan05] CVSscan: Visualization of Code Evolution - Lucian Voinea, Alex Telea, Jarke J. van Wijk - 2005
- [CodeCrawler] - A Lightweight Software Visualization Tool - Michele Lanza
- [Cowan94] Enhancing Code for Readability and Comprehension Using SGML - D.D. Cowan, D.M. Germin, C.J.P. Lucena, A. von Staa - 1994
- [Craw91] SID: a graphical browser – Tamil L. Crawford - Proceedings of the 1991 ACM SIGSMALL/PC symposium on Small systems
- [CSS] Hakon Wium Lie and Bert Bos - Cascading Style Sheets, Addison-Wesley, 1997.
- CodeCrawler - A Lightweight Software Visualization Tool - Michele Lanza
- [Domin97] Staging Software Visualizations on the Web – John Dominigue, Paul Mulholland
- [Esdt88] Eisenstadt, M. and M. Brayshaw (1988). The Transparent Prolog Machine (TPM): An Execution Model and Graphical Debugger for Logic Programming.
- [Gamma95] Design Patterns: Elements of Reusable Object-oriented Software - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides – 1995.
- [Garrett05] Ajax: A New Approach to Web Applications - JJ Garrett - Adaptive Path, 2005
www.adaptivepath.com/publications/essays/archives/000385.php
- [Gears] Google Gears - <http://gears.google.com/>
- [Goldstein47] Goldstein, H.H. and I. von Neumann (1947). Planning and Coding Problems for an Electronic Computing Instrument. Reprinted in von Neumann, J.. Collected Work.. A.H. Taub. (Ed.) New York McMillan. 80-151.

[Haibt59] Haibt, L.M. (1959). A Program to Draw Multi-Level Flow Charts. Proc. of The Western Joint Computer Conf., 131-157.

[Jackobsen] Evaluating a Fisheye View of Source Code - Mikkel Rønne Jakobsen & Kasper Hornbæk – 2006

[Jern98] “Thin” vs. “fat” visualization client – Jern, M. - 1998

[Knuth84] Knuth, D. E. (1984). Literate Programming. *The Computer Journal*, 27(2): 97-111.

[Ledgard77] Hueras, J. & Ledgard, H. (1977). An Automatic Formatting Program for Pascal. *ACM SIGPLAN Notices*, 12(7): 82-84.

[Loosley06] Rich Internet Applications: Design, Measurement, and Management Challenges, - Chris Loosley - Keynote Systems, 2006

[Mayrh95] Program comprehension during software maintenance and evolution - Von Mayrhauser, A. Vans, A.M. - 1995

[MDC] Introduction to Object-Oriented JavaScript – *Mozilla Developer Center*
http://developer.mozilla.org/en/docs/Introduction_to_Object-Oriented_JavaScript

[Miller] Miller, George, "The Magic Number Seven Plus or Minus Two: Some Limits on Our Capacity for Processing Information", *Psychological Review*, 63,81-97.

[Myers86] Myers, Brad, "Visual Programming, Programming by Example, and Program Visualization: A Taxonomy", CH1 '86 Proceedings, 1986

[Pareja03] Program execution and visualization on the web - Web-based education: learning from experience - C. Pareja-Flores, J. Á. Velázquez-Iturbide - 2003

[Paulson05] Building Rich Web Applications with Ajax – LD Paulson - IEEE Computer Society Press - 2005

[Pike] Interpreting the Data Parallel Analysis with Sawzall - Rob Pike, Sean Dorward, Robert Griesemer, Sean Quinlan - Google, Inc.

[_Prec] Designing Rich Internet Applications with Web Engineering Methodologies - J. C. Preciadol; M. Linajel; S. Comai2; F. Sanchez-Figueroal

[Price92] Price, B. A., Small, I. S., & Baecker, R. M. - A Taxonomy of Software Visualization - 1992

[Price93] A Principled Taxonomy of Software Visualization, by Blaine A. Price +*, Ronald M. Baecker *, and Ian S. Small - *Journal of Visual Languages and Computing*, 4(3), 1993, pp. 211-266.

[Proactor] Proactor: An Object Behavioral Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events - *Irfan Pyarali, Tim Harrison, and Douglas C. Schmidt, Thomas D. Jordan*

[Rigi] Tilley, S.R., Wong, K., Storey M.A.D., Muller, H.A. - Rigi: A visual tool for understanding legacy systems - In International Journal of Software Engineering and Knowledge Engineering, - December 1994

[Roberts06] An interactive tutorial system for Java – Eric Roberts – Stanford University – 2006

[Shneiderman98] Designing the user interface: strategies for effective human-computer interaction – B. Shneiderman - 1998

[SHriMP] Storey, M.A., Best, C., Michaud, J., Rayside, D., Litoiu, M., Musen, M - SHriMP Views: an Interactive Environment for Information Visualization and Navigation - Proc. CHI '02, ACM Press, NY, 520 – 521.

[Standish84] Standish, T.A. An Essay on Software Reuse. *IEEE Trans. On Software Engineering*, 10 (5), Sep. 1984, 494 — 497.

[Stasko90] Stasko, J. T. - Tango: A Framework and System for Algorithm Animation - *IEEE Computer*, 23(9): 27-39 - 1990

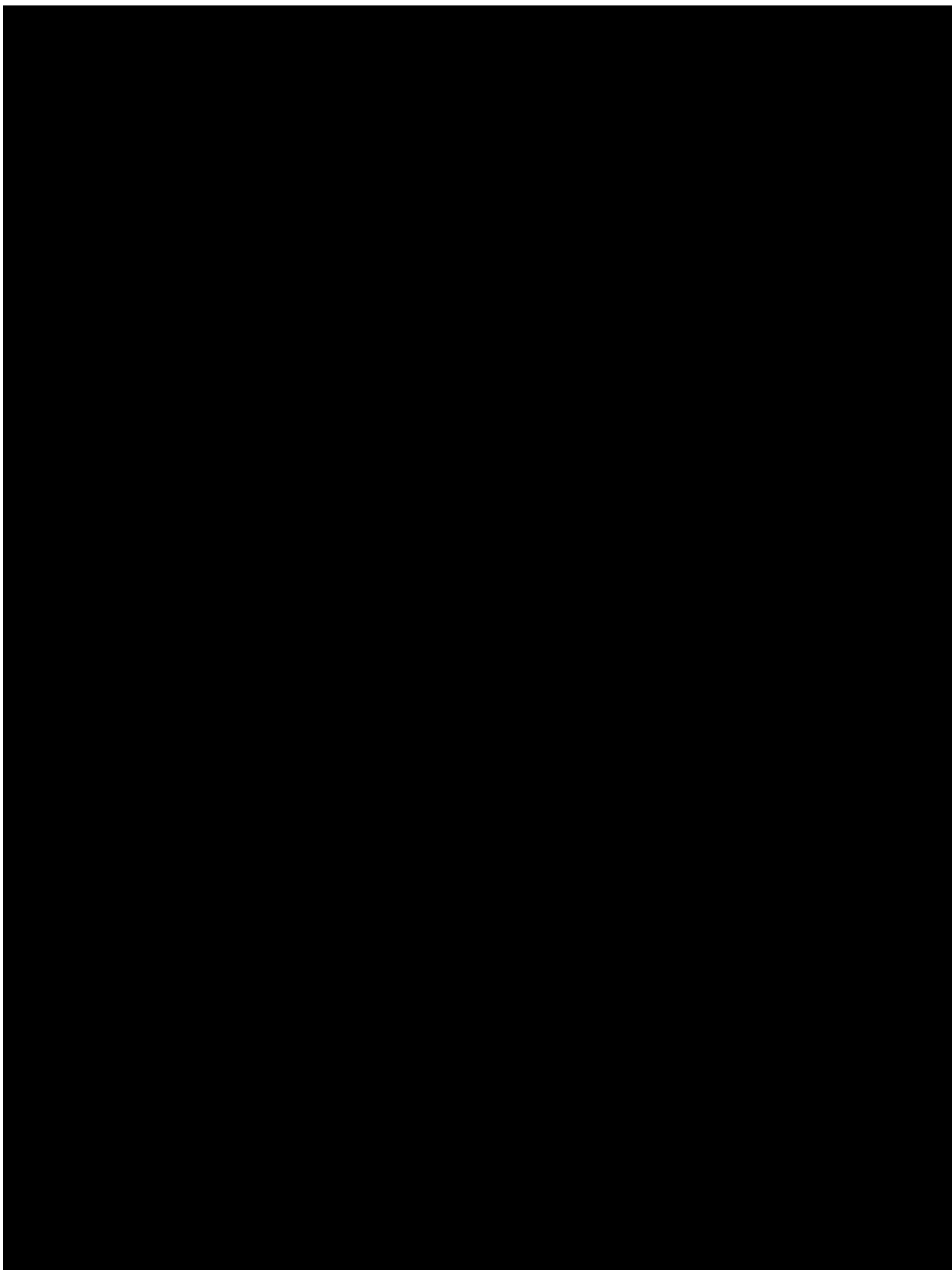
[Stasko98] Software Visualization - John T. Stasko, John B. Domingue, Marc H. Brown and Blaine A. Price – 1998

[SWATO] The SWATO JavaScript Template Engine – <http://swato.dev.java.net/>.

[Tilley97] On Using the Web as Infrastructure for Reengineering – Scott R. Tilley Carnegie, Mellon University

[Yu06] XUPClient - A Thin Client for Rich Internet Applications - Jin Yu, Boualem Benatallah, Fabio Casati, and Regis Saint-Paul - 2006

Appendix A.



Appendix B. Usage statistics for GCL Viewer

In figure B.1 we show statistics on the number of visits and visitors to GCL Viewer over the period Jan 10, 2008 – Jun 17, 2008. The initial session by a user during any given date range is considered to be an additional *visit* and an additional *visitor*. Any future sessions from the same user during the selected time period are counted as additional *visits*, but not as additional *visitors*.

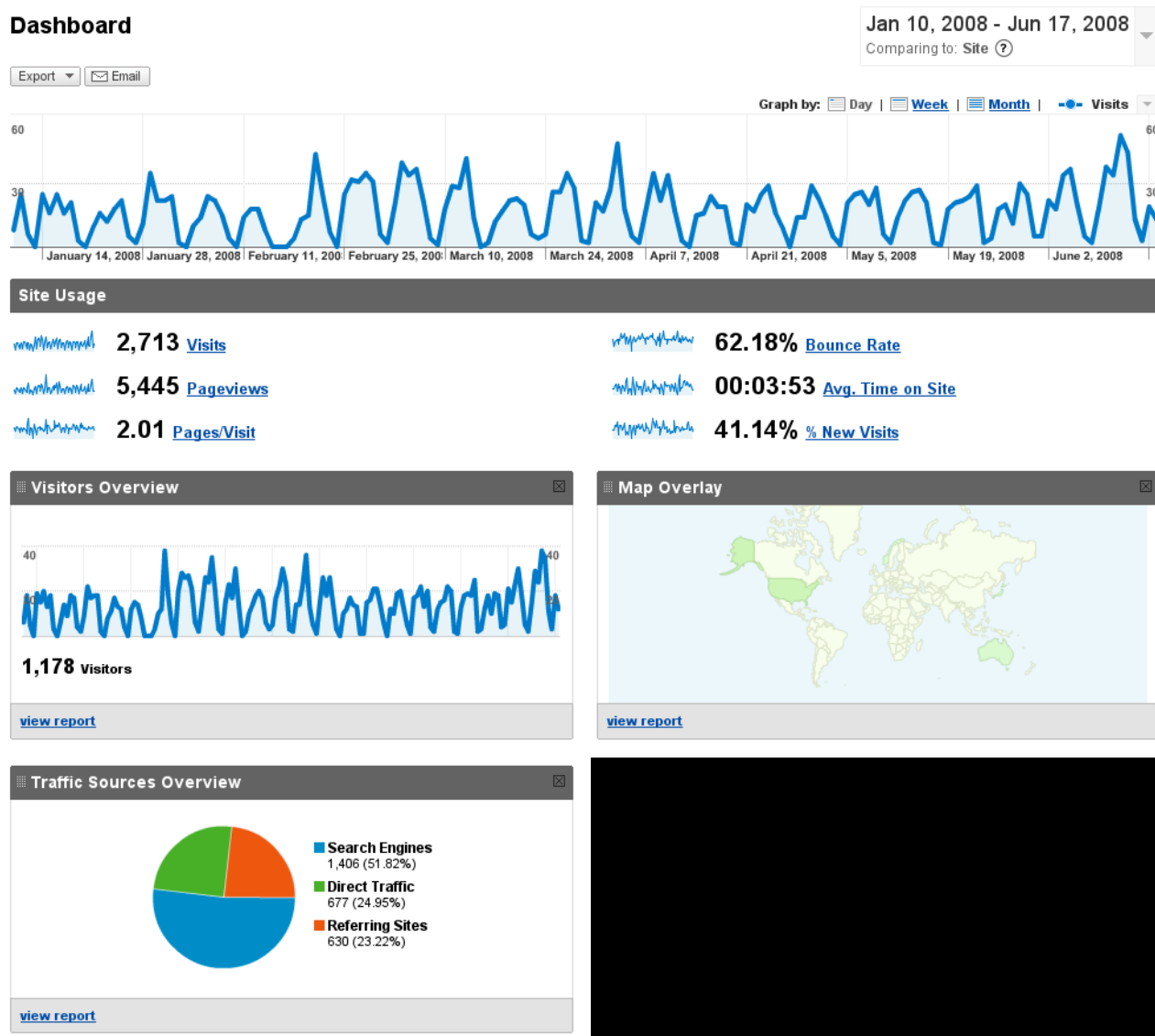


Figure B.1: Number of visits and visitors over the period Jan 10, 2008 – Jun 17, 2008

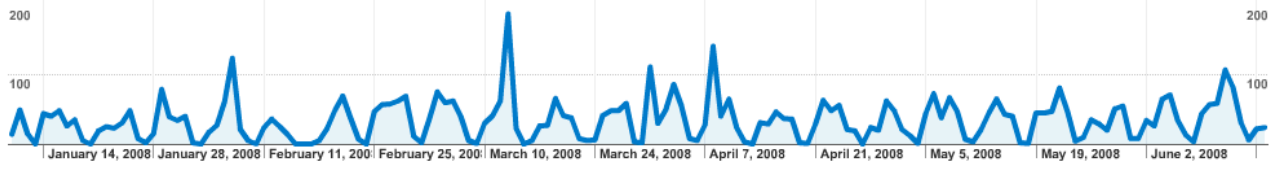
In figure B.2 we show the number of page views that were generated on GCL Viewer over the period Jan 10, 2008 – Jun 17, 2008. Page views is the total number of pages viewed on a site and is a general measure of how much a site is used. In our case it is a good measure of the number of GCL programs that have been viewed (viewing a GCL program generates a new page).

Dashboard

Jan 10, 2008 - Jun 17, 2008
Comparing to: Site ?

Export Email

Graph by: Day | Week | Month | Pageviews



Appendix C. Feedback from GCL Viewer users

In this appendix we list some of the many positive feedback that we have received from GCL Viewer users. We present two groups of feedback: feedback that was received spontaneously over email and feedback from the survey study.

C.1 Spontaneous feedback via email

- *“Thank you guys for creating such a useful tool, just when I was looking for something like this.”*
- *“I can finally make sense of these randomly constructed GCL files...”*
- *“GCL Viewer rocks. I was going crazy before someone told me about it :)”*
- *“The new dependency coloring is very cool :)”*

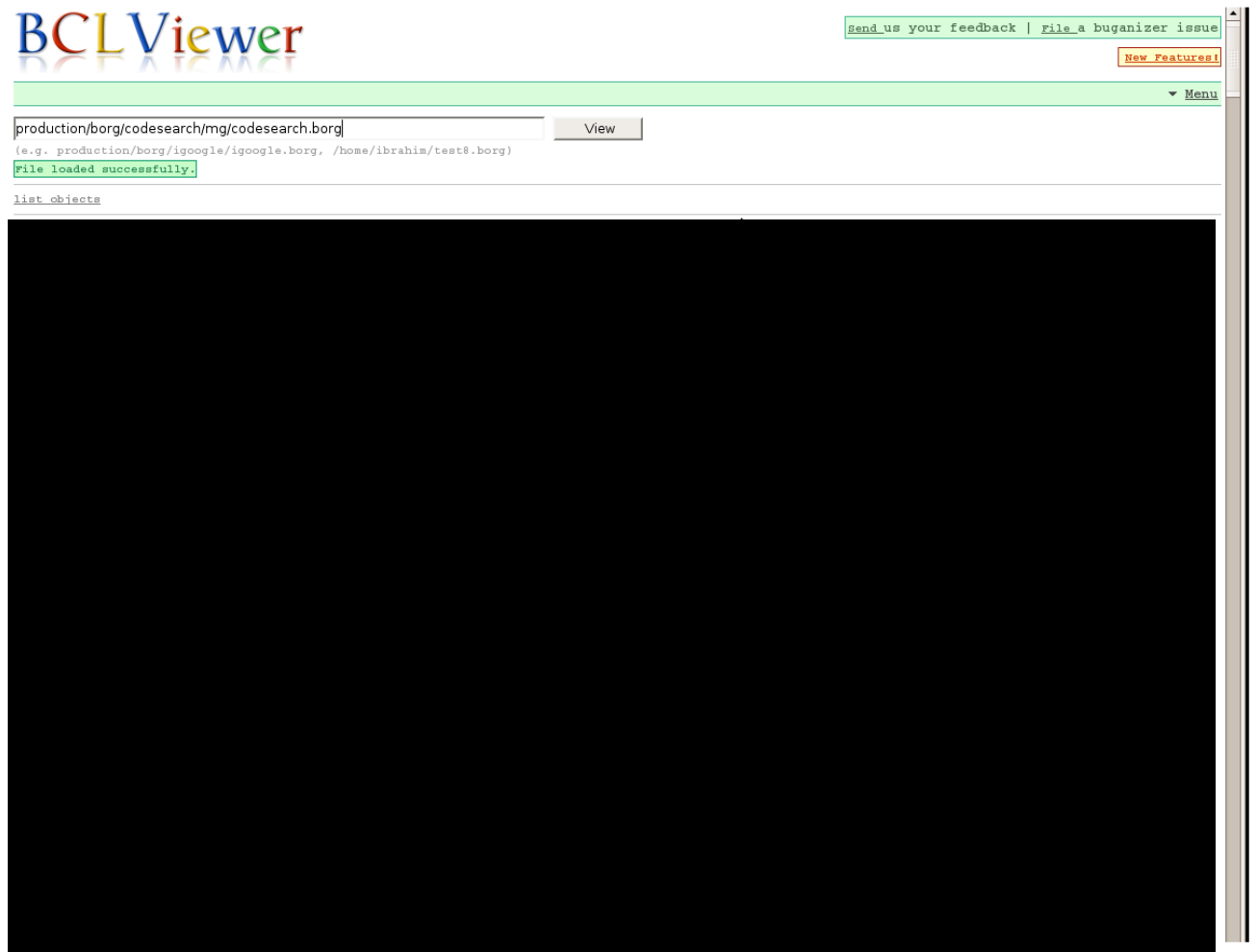
C.2 Feedback via survey

- *“GCL Viewer is one of the best and most-needed tools at Google IMHO. “*
- *“This is a great tool. I use it to quickly expand, evaluate and validate my gcl files. It saves me time since it does automatically what I should do manually in order to track down issues.”*
- *“Awesome tool. Thanks. Very useful for code reviews BTW.”*
- *“I love it.”*
- *“Excellent tool”*

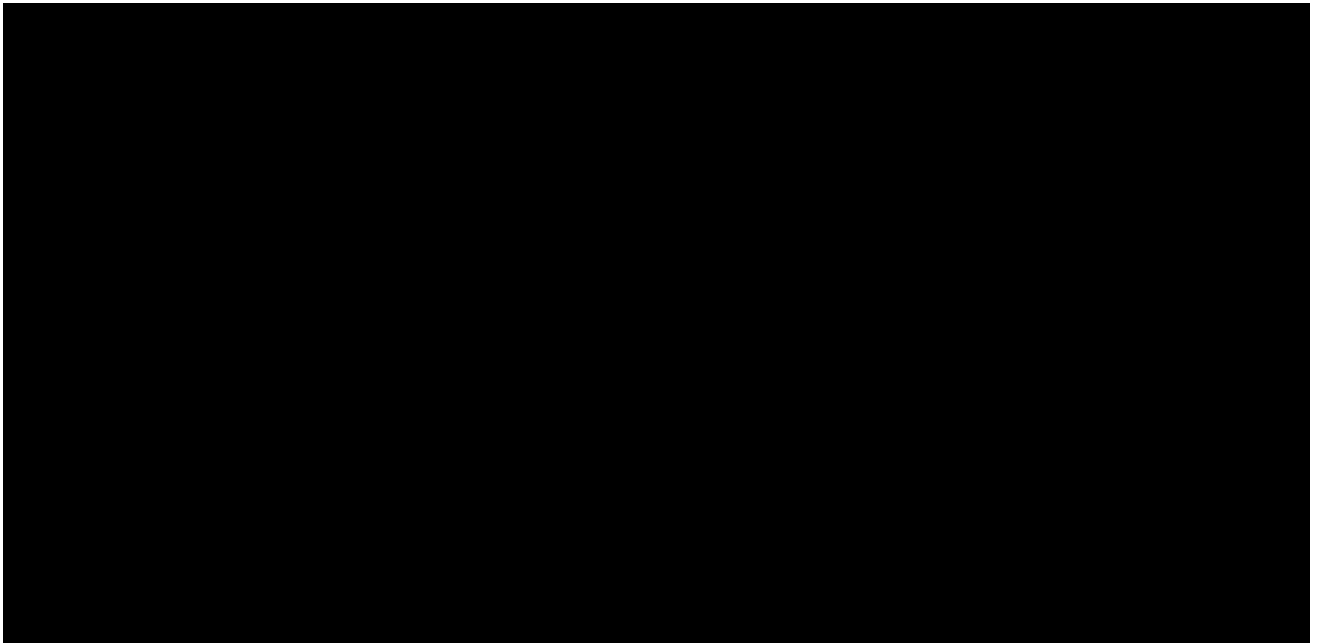
Appendix D. Screenshots

In this appendix we provide screenshot to show some of the features of GCL Viewer.

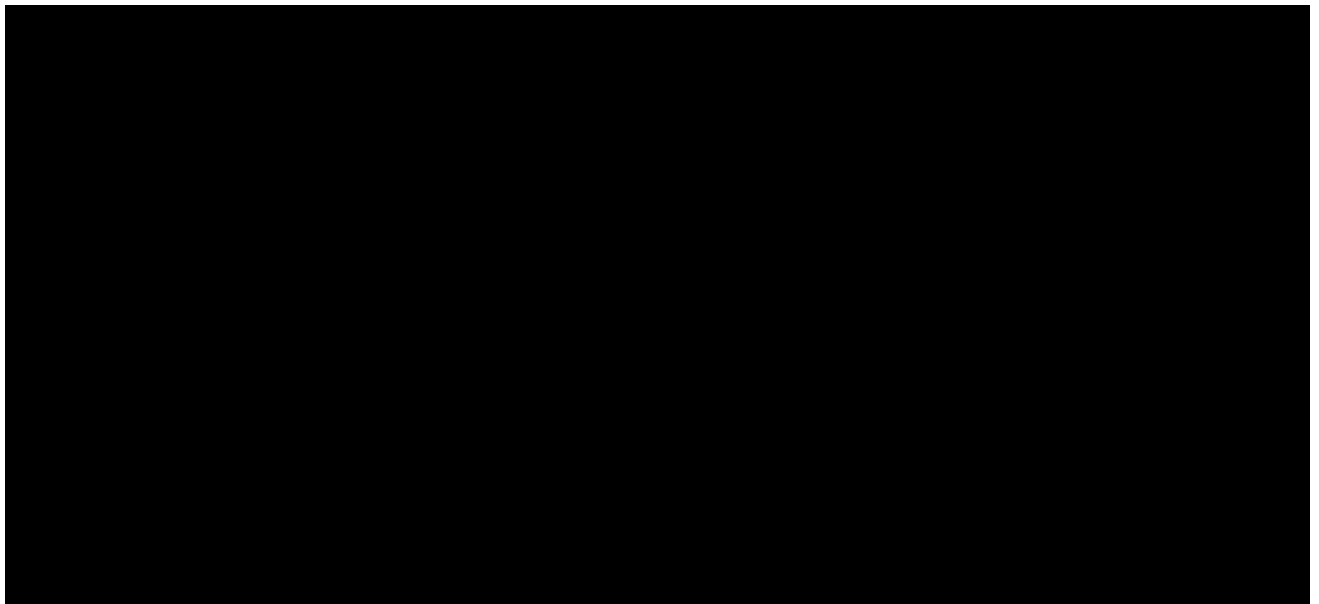
1. General



2. Dependency Coloring



3. Error messages and assertions



4. Field Inspector

