Chapter 6

Case Studies: Realizing the Infrastructure with g^4re

In this chapter we present two case studies in which we use the g^4re tool chain to realize our infrastructure. We designed our case studies to determine the space and time costs incurred by the use of our infrastructure. We measure space in two dimensions: size on disk, and size of graph(s), i.e., the number of nodes and edges. We measure times for parsing and building in-memory representations, as well as for the linking process, and the application of XSLT style sheets.

First, in Section 6.1, we describe the twelve applications and libraries that serve as the test suite in our case studies. In Section 6.2, we exchange low-level graphs, and measure the space and time costs incurred. In Section 6.3, we exchange middle-level graphs, and again measure the space and time costs incurred. In this section we also apply XSLT style sheets to each middle-level graph. We use style sheets that summarize the contents of each middle-level graph instance; the process of writing the style sheets, which requires knowledge of only the schema, is automatable.

6.1 Test Suite

In Table 6.1, we list the twelve open source applications and libraries, or test cases, that form the test suite for our studies[1]. In the first column, we list the names that we use to refer to the test cases. In the next three columns of the table, we list relevant data about the test cases. We list the version numbers in the second column, the number of C++ translation units in the third, and the approximate number of thousands of lines of non-commented, non-preprocessed lines of code in the fourth.

The twelve applications and libraries that form our test suite are widely used, are freely available on the Web, and consist of approximately one million lines of non-commented, non-

[1] Additional information about each test case is available in our online repository.

Test Case	Version	C++ Translation Units	NCLOC (≈ K)
AvP	CVS 07/22/05	95	295
CppUnit	1.10.2	51	4
Doxygen	1.4.4	69	170
FluxBox	0.9.14	107	32
FOX	1.4.17	245	110
HippoDraw	1.15.8	249	55
Jikes	1.22	38	70
Keystone	0.2.3	52	16
Licq	1.3.0	28	36
Pixie	1.5.2	78	80
Scintilla	1.66	78	35
Scribus	1.2.3	110	80

Table 6.1: Test suite. *The 12 test cases that we use in our study. For each test case, we list the version, the number of C++ translation units, and the approximate number of thousands of non-commented, non-preprocessed lines of code (NCLOC). The test suite contains 1,200 C++ translation units and approximately one million lines of code.*

preprocessed code. *AvP* is a Linux port of the Fox Interactive/Rebellion Developments game Aliens vs Predator (Gold Edition) [Rebellion 2005]. *CppUnit* is a C++ port of the JUnit framework for unit testing [CppUnit Project 2006]. *Doxygen* is a documentation system for C, C++, and Java [van Heesch 2006]. *FluxBox* is a light-weight X11 window manager built for speed and flexibility [FluxBox Project 2006]. *FOX* is a toolkit to facilitate development of graphical user interfaces [van der Zijp 2006]. *HippoDraw* provides a highly interactive data analysis environment [Kunz 2006]. *Jikes* is a Java compiler system from IBM [IBM Jikes Project 2006]. *Keystone* is a parser and front end for ISO C++ [Keystone Project 2005; Malloy et al. 2003a]. *Licq* is a multi-threaded ICQ clone [Licq Project 2006]. *Pixie* is a RenderMan ® like photorealistic renderer [Arikan 2006]. *Scintilla* is a source code editing component that includes support for syntax styling, error indicators, code completion, and call tips [Hodgson 2006]. The final test case, *Scribus*, is a professional, cross-platform desktop publishing system [Scribus Project 2006].

We executed all experiments on a custom workstation with a *Dual Core AMD Opteron* ^{TM} *165* processor, 2048 MB of PC3200 DDR RAM, and a 250 GB, 7200 RPM SATA II hard disc on which we installed the Slackware 10.2 operating system after formatting with version 3.6 of the *ReiserFS* filesystem. We performed the experiments with version 1.5.0 of g^{4}re, which we compiled with version 4.1.1 of *gcc*. We created all `tu` files with *gcc* version 3.3.6.

6.2 Case Study: Exchanging Low-Level Graphs

In this section we describe the results of our first case study, in which we examine low-level graphs from our infrastructure. g^{4}re exchanges multiple formats, as discussed in Subsection 5.2.1. In Subsections 6.2.1 and 6.2.2, we describe the formats that g^{4}re exchanges, and present results for exchanging GXL encoded instances of schemas at Level 0 and I of our infrastructure, respectively. We discuss the results of the case study in Subsection 6.2.3.

6.2.1 Exchanging Graphs at Level 0

In this subsection we investigate the costs associated with exchanging instances of low-level graphs; in particular, we investigate the costs of exchanging instances of the GENERIC ASG schema, in both `tu` and GXL formats. First, we illustrate the two exchange formats. Second, we measure the space and time costs incurred by exchanging ASGs, which are found in Level 0 of our infrastructure.

```
1 class Base { };
2 class Parser : public Base { };
```

Source Listing 6.1: Source code for class Parser. *Definition of the C++ class Parser. Parser inherits from the class Base.*

In Source Listing 6.1, we list C++ code for the definition of class `Parser`. We list a base class, `Base`, on line 1, and the class `Parser` on line 2. The inheritance relationship between `Parser` and `Base` is public and non-virtual.

We list the definition of a C++ class, `Parser`, in the GENERIC `tu` file format in Source Listing 6.2, and the corresponding definition as a GXL encoded instance of the GENERIC schema in Source Listing 6.3. GXL is clearly more verbose than the *gcc* `tu` file format; the respective character counts for the text in the figures are 447 and 1178.

```
@3      type_decl          name: @4        type: @5        srcp: Parser.cpp:2
                           artificial      chan: @6        addr: b7e0a460
@4      identifier_node    strg: Parser    lngt: 6         addr: b66b3ac0
@5      record_type        name: @3        size: @7        algn: 8
                           base: @8        public          struct
                           flds: @9        fncs: @10       binf: @11
                           addr: b7e0a310
```

Source Listing 6.2: Instance of a `tu` file. *Definition of class Parser as represented in a tu file. A node definition in a tu file consists of: a unique integer prepended with "@", a string representing the node type, edges of the form "edge: dest", fields of the form "field: value", and a set of single word attributes.*

```
<node id="n3">
  <type xlink:href="GENERIC.gxl#type_decl"/>
  <attr name="attr"><set><string>artificial</string></set></attr>
  <attr name="srcp"><string>Parser.cpp:2</string></attr>
</node>
<edge from="n3" to="n4"><type xlink:href="GENERIC.gxl#name"/></edge>
<edge from="n3" to="n5"><type xlink:href="GENERIC.gxl#type"/></edge>
<edge from="n3" to="n6"><type xlink:href="GENERIC.gxl#chan"/></edge>
<node id="n4">
   <type xlink:href="GENERIC.gxl#identifier_node"/>
   <attr name="attr"><set></set></attr>
   <attr name="strg"><string>Parser</string></attr>
</node>
<node id="n5">
   <type xlink:href="GENERIC.gxl#record_type"/>
   <attr name="attr"><set><string>struct</string></set></attr>
   <attr name="qual"><string></string></attr>
</node>
<edge from="n5" to="n8">
   <type xlink:href="GENERIC.gxl#base"/>
   <attr name="base"><tup><bool>false</bool><string>public</string></tup></attr>
</edge>
<edge from="n5" to="n3"><type xlink:href="GENERIC.gxl#name"/></edge>
<edge from="n5" to="n7"><type xlink:href="GENERIC.gxl#size"/></edge>
<edge from="n5" to="n10"><type xlink:href="GENERIC.gxl#fncs"/></edge>
<edge from="n5" to="n11"><type xlink:href="GENERIC.gxl#binf"/></edge>
```

Source Listing 6.3: GXL instance of the GENERIC schema. *Definition of class Parser as represented in a GXL encoded instance of the* GENERIC *schema. The* GENERIC *GXL schema is a direct encoding of the tu file format, but with internal gcc information, such as addresses and string lengths, omitted. The "@" symbol is translated to "n" to conform to XML standards.*

	.cpp.tu[.gxl][.gz]	.cpp.tu[.gz]	.cpp.tu.gxl[.gz]
Test Case	Nodes	Edges	Edges
AvP	3 286 604	8 607 856	8 509 901
CppUnit	4 574 861	10 983 481	10 911 237
Doxygen	7 558 527	17 894 321	17 724 872
FluxBox	12 016 093	30 111 171	29 852 859
FOX	12 139 219	32 260 488	31 953 355
HippoDraw	18 835 420	44 662 239	44 338 296
Jikes	7 543 803	17 437 798	17 321 098
Keystone	6 159 791	15 152 153	15 047 146
Licq	2 663 307	6 813 822	6 751 433
Pixie	3 278 791	7 665 603	7 620 166
Scintilla	1 414 562	3 456 874	3 427 785
Scribus	17 418 294	44 859 563	44 426 635

Table 6.2: Level 0: Numbers of nodes and edges. *The numbers of nodes and edges for ASGs that represent the test cases.*

Note that the text in Source Listing 6.2 contains information not present in Source Listing 6.3. Extraneous information, such as an address or string length, is omitted from the GXL encoding. Empty lists are detected and removed during encoding; the `flds` edge is omitted in this example. The `fncs` edge is not omitted, because *gcc* provides a constructor, copy constructor, and assignment operator for each class.

It is well known that XML imposes significant storage costs; however, this fact has not hindered its wide spread adoption. Due to the prevalence of XML, there are several tools, available in popular languages such as C, C++, and Java, that were designed with these costs in mind. We designed and implemented a wrapper for the XML parser *expat* [eXpat Project 2005] that uses *zlib* [zlib Project 2005] to read compressed files. We also implemented a subclass of the C++ standard library class `ostream` to write compressed files. To provide a complete comparison, we instrumented our *flex* scanner to read compressed `tu` files.

In Table 6.2, we list the numbers of nodes and edges for ASGs that represent the test cases. In column 1, we list the test cases. In column 2, we list the number of nodes in the possibly GXL-encoded instance of the GENERIC schema. In columns 3 and 4, we list the numbers of edges in the `tu` files and GXL encoded `tu` files, respectively.

We apply the pruning algorithm discussed in Subsection 5.1.1 during the parse of a `tu`

Test Case	.cpp.tu[.gz]		.cpp.tu.gxl[.gz]	
AvP	809	84	1 376	122
CppUnit	567	98	1 784	116
Doxygen	863	152	2 794	172
FluxBox	1 540	250	4 842	312
FOX	1 643	230	5 162	303
HippoDraw	2 283	376	7 222	469
Jikes	872	145	2 795	181
Keystone	773	126	2 439	157
Licq	341	56	1 081	69
Pixie	414	56	1 202	71
Scintilla	177	27	554	34
Scribus	2 184	352	6 967	440

Table 6.3: Level 0: Size on disk (MB). *The size on disk, in megabytes, for ASGs that represent the test cases.*

file. We show the effects of our pruning algorithm in Table 6.2. Our pruning algorithm does not remove any nodes, but it does remove edges. In the table, we show the difference between the numbers of edges in the `tu` files and the GXL encodings of the `tu` files. Next, we investigate the storage costs introduced by the use of GXL, and the saving that can be achieved by compressing files of each format.

In Table 6.3, we list the sizes on disk, in megabytes, for ASGs that represent the test cases. In column 1, we list the test cases. In columns 2 and 3, we list the total size of the uncompressed and compressed tu files, respectively. In columns 4 and 5, we list the total sizes of the uncompressed and compressed GXL encoded tu files, respectively.

A comparison of columns 2 and 4 of the table shows the significant storage cost introduced by the use of uncompressed GXL. For all but one of the test cases, the uncompressed GXL encodings of the `tu` files more than double the storage costs. For example, the total storage cost of the `tu` files for *Jikes* is 872 megabytes, but the total storage cost of the GXL encodings is 2 795 megabytes; the `tu` files are 3.2 times smaller than the GXL encodings. The outlier is *AvP*, for which the `tu` files, at 809 megabytes, are only 1.7 times smaller than the GXL encodings. On average, uncompressed `tu` files are 3.02 times smaller than the GXL encodings of the `tu` files, with a standard deviation of 0.42. Columns 3 and 5

Test Case	.cpp.tu[.gz]		.cpp.tu.gxl[.gz]	
AvP	97.39	112.62	136.58	155.71
CppUnit	123.47	142.85	174.66	199.56
Doxygen	206.07	238.65	279.39	322.82
FluxBox	341.50	388.24	472.39	552.80
FOX	347.15	411.66	503.60	577.47
HippoDraw	514.72	584.73	715.28	829.80
Jikes	208.93	233.54	253.77	291.35
Keystone	171.89	194.75	239.23	275.83
Licq	76.06	87.47	90.77	105.75
Pixie	86.74	104.06	125.20	144.57
Scintilla	38.63	46.65	56.30	64.99
Scribus	508.73	572.60	600.87	703.67

Table 6.4: Level 0: Time (s). *The running time, in seconds, to parse and build in-memory representations of ASGs that represent the test cases.*

show the significant savings in storage cost that compression introduces when compared to columns 2 and 4, respectively. In addition, the gap between the storage costs of the two file formats is significantly reduced when compression is used. On average, compressed `tu` files are 1.25 times smaller than the GXL encodings of the `tu` files, with a standard deviation of 0.08. GXL, and XML in general, compresses at a higher ratio than other text formats. Next, we investigate the run-time costs introduced by the use of compression and GXL.

In Table 6.4, we list the running times, in seconds, to parse and build in-memory representations of ASGs that represent the test cases. In column 1, we list the test cases. In columns 2 and 3, we list the total times for the uncompressed and compressed tu files, respectively. In columns 4 and 5, we list the total times for the uncompressed and compressed GXL encoded tu files, respectively.

As stated in Subsection 5.1.1, we parse `tu` files using a *flex* generated scanner, GXL files using *expat*, and compressed files using *zlib*. We use the same node list graph data structure to store each graph instance in memory. A comparison of columns 2 and 4 of the table shows the run-time cost introduced by the use of GXL. The running times for GXL inputs are consistently higher than those for `tu` inputs, but the run-time costs introduced by GXL are much lower than the corresponding storage costs. On average, parsing the uncompressed `tu` files is 1.36 times faster than parsing the uncompressed GXL encodings of the `tu` files, with a standard deviation of 0.10. The average time for uncompressed `tu` files is 226.77 seconds, with a standard deviation of 164.86. On average, parsing the compressed `tu` files is 1.36 times faster than parsing the compressed GXL encodings of the `tu` files, with a standard deviation of 0.08. The average time for compressed `tu` files is 259.82 seconds, with a standard deviation of 186.81.

6.2.2 Exchanging Graphs at Level I

In this subsection we continue to investigate the costs associated with exchanging instances of low-level graphs; in particular, we investigate the costs of exchanging instances of the CppInfo schema. First, we illustrate a GXL encoded instance of the CppInfo schema. Second, we measure the space and time costs incurred by exchanging APIs, which are found in Level I of our infrastructure.

We list the definition of C++ class `Parser` (see Source Listing 6.1 for details) as a GXL encoded, linked instance of the CppInfo schema in Source Listing 6.4. The character count for the text in the figure is 1307, which is larger than even the GXL encoding of the original `tu` file. However, we implemented maximal sharing of strings, such as file and identifier names, and integers, such as line and column numbers, to improve the scalability of this format.

We show the effects of our linking process in Table 6.5. In the table, we show the differences between the numbers of nodes and edges in the intermediate (unlinked) instances and the linked instances of the CppInfo schema. In columns 2 and 3, we list the sums of nodes and edges, respectively, for all intermediate instances for each test case. The numbers of nodes and edges for intermediate instances vary widely. The minimum number of nodes is 780 024 for *Scintilla*, and the maximum number of nodes is 10 164 005 for *HippoDraw*. The minimum number of edges is 2 391 321 for *Scintilla*, and the maximum number of edges is 34 941 134 for *HippoDraw*. The average numbers of nodes and edges are 4 262 119 and 14 445 413, with standard deviations of 3 402 982 and 11 469 128, respectively.

```
<node id="n781">
  <type xlink:href="CppInfo.gxl#ClassNonTemplate"/>
  <attr name="visibility"><enum></enum></attr>
  <attr name="isConst"><bool>false</bool></attr>
  <attr name="isVolatile"><bool>false</bool></attr>
  <attr name="key"><enum>class</enum></attr>
</node>
<edge from="n781" to="n782"><type xlink:href="CppInfo.gxl#HasSourceLocation"/></edge>
<edge from="n781" to="n1"><type xlink:href="CppInfo.gxl#HasScope"/></edge>
<edge from="n781" to="n784"><type xlink:href="CppInfo.gxl#HasName"/></edge>
<edge from="n781" to="n758" toorder="24">
  <type xlink:href="CppInfo.gxl#Bases"/>
  <attr name="inheritanceSpecifier">
    <tup><enum>public</enum><bool>false</bool></tup>
  </attr>
</edge>
<edge from="n781" to="n785" toorder="28">
  <type xlink:href="CppInfo.gxl#Functions"/>
</edge>
<edge from="n781" to="n790" toorder="29">
  <type xlink:href="CppInfo.gxl#Functions"/>
</edge>
<edge from="n781" to="n795" toorder="30">
  <type xlink:href="CppInfo.gxl#Functions"/>
</edge>
<node id="n782">
  <type xlink:href="CppInfo.gxl#SourceLocation"/>
</node>
<edge from="n782" to="n760"><type xlink:href="CppInfo.gxl#HasFilename"/></edge>
<edge from="n782" to="n783"><type xlink:href="CppInfo.gxl#HasLine"/></edge>
<edge from="n782" to="n4"><type xlink:href="CppInfo.gxl#HasColumn"/></edge>
<node id="n783">
  <type xlink:href="CppInfo.gxl#SourcePosition"/>
  <attr name="number"><int>2</int></attr>
</node>
<node id="n784">
  <type xlink:href="CppInfo.gxl#Identifier"/>
  <attr name="string"><string>Parser</string></attr>
</node>
```

Source Listing 6.4: GXL instance of the CppInfo schema. *Definition of class Parser as represented in the GXL encoded, linked instance of the* CppInfo *schema.*

Test Case	.cpp.tu.ci.gxl[.gz] Nodes	Edges	.cil.gxl[.gz] Nodes	Edges
AvP	2 059 850	6 321 574	148 972	631 882
CppUnit	2 657 601	9 208 857	85 355	330 845
Doxygen	2 234 210	7 956 801	208 463	805 926
FluxBox	6 562 227	23 026 116	215 846	1 264 464
FOX	9 631 093	29 647 216	221 383	1 016 806
HippoDraw	10 164 005	34 941 134	254 270	1 470 270
Jikes	2 932 380	10 204 160	154 132	554 202
Keystone	3 314 379	11 731 213	139 570	625 173
Licq	1 142 403	3 996 935	128 045	541 960
Pixie	1 538 147	4 832 153	109 408	491 839
Scintilla	780 024	2 391 321	129 658	437 110
Scribus	8 129 110	29 087 482	330 537	1 510 133

Table 6.5: Level I: Numbers of nodes and edges. *The numbers of nodes and edges for APIs that represent the test cases.*

In columns 4 and 5 of Table 6.5, we list the numbers of nodes and edges, respectively, for the linked instance for each test case. These numbers are substantially smaller than those for the intermediate instances. The minimum number of nodes is 177 355 for *CppUnit*, and the maximum number of nodes is 254 270 for *HippoDraw*. The minimum number of edges is 330 845 for *CppUnit*, and the maximum number of edges is 1 470 270 for *HippoDraw*. The average numbers of nodes and edges are 177 136 and 806 717, with standard deviations of 70 277 and 409 918, respectively. The substantial reductions indicate a high ratio of duplication among translation units for all test cases. Recall that duplication is the result of compiler-specific information, as well as header files, being present in multiple translation units. Next, we investigate the savings in storage costs introduced by the linking process.

In Table 6.6, we list the sizes on disk, in megabytes, for APIs that represent the test cases. In column 1, we list the test cases. In columns 2 and 3, we list the total size of the uncompressed and compressed GXL encoded, intermediate instances of the CppInfo schema, respectively. In columns 4 and 5, we list the total sizes of the uncompressed and compressed GXL encoded, linked instances of the CppInfo schema, respectively.

A comparison of columns 2 and 3 of the table to columns 4 and 5 of the table, respectively, shows the significant savings introduced by the linking process. For all test cases, the

Test Case	.cpp.tu.ci.gxl[.gz]		.cil.gxl[.gz]	
AvP	1 586	62	99	5
CppUnit	3 443	103	54	3
Doxygen	2 102	80	126	7
FluxBox	8 609	258	188	10
FOX	7 270	279	149	8
HippoDraw	12 826	389	219	11
Jikes	3 425	111	89	5
Keystone	4 404	132	98	5
Licq	1 380	44	85	5
Pixie	1 212	47	73	4
Scintilla	625	24	71	4
Scribus	7 932	289	231	12

Table 6.6: Level I: Size on disk (MB). *The size on disk, in megabytes, for APIs that represent the test cases.*

uncompressed GXL encoding of the linked instance is at least 8.8 times smaller than the uncompressed GXL encodings of the intermediate instances. For example, the total storage cost of the linked instance for *Jikes* is 89 megabytes, but the total storage cost of the intermediate instances is 3 425 megabytes; the linked instance is 38.5 times smaller than the intermediate instances. *CppUnit* shows the biggest difference in storage costs, with the linked instance 63.8 times smaller than the intermediate instances. *Scintilla* shows the smallest difference in storage costs. The savings for the compressed GXL encodings are similar, although the ratios drop slightly due to the high rate of compression. A large reduction in size indicates a high level of duplication among translation units (intermediate instances), likely caused by poor compiler firewalling. Next, we investigate the savings in run-time costs introduced by the linking process.

In Table 6.7, we list the running times, in seconds, to parse and build in-memory representations of APIs that represent the test cases. In column 1, we list the test cases. In columns 2 and 3, we list the total times for the uncompressed and compressed GXL encoded, intermediate instances of the CppInfo schema, respectively. In columns 4 and 5, we list the total times for the uncompressed and compressed GXL encoded, linked instances of the CppInfo schema, respectively.

Test Case	.cpp.tu.ci.gxl[.gz]		.cil.gxl[.gz]	
AvP	110.81	116.17	8.43	9.47
CppUnit	202.19	217.50	4.70	5.19
Doxygen	143.85	150.62	10.89	11.53
FluxBox	521.57	548.98	16.01	16.52
FOX	516.11	542.16	12.46	13.56
HippoDraw	774.65	815.89	18.23	20.29
Jikes	211.79	223.77	7.68	8.12
Keystone	264.85	270.78	8.84	9.05
Licq	85.50	88.86	7.51	7.75
Pixie	83.88	87.96	6.10	6.52
Scintilla	42.72	44.70	6.08	6.58
Scribus	534.36	445.43	19.46	21.58

Table 6.7: Level I: Time (s). *The running time, in seconds, to parse and build in-memory representations of APIs that represent the test cases.*

A comparison of columns 2 and 4 shows a significant savings in run-time costs when dealing with a linked representation of a program. This result follows directly from the significant savings in storage costs shown in Tables 6.5 and 6.6. The time to parse a linked instance is well under 30 seconds for all test cases, whether or not the GXL encoding is compressed. The time to parse the intermediate instances is under 60 seconds for only one test case, and over half of the test cases take over three minutes to parse. The maximum time to parse compressed GXL encodings of intermediate instances is nearly 15 minutes, for *HippoDraw*.

6.2.3 Discussion

The results for exchanging low-level graphs show that the storage costs can be prohibitive. The largest files recorded in this case study, uncompressed GXL encodings of intermediate instances of the CppInfo schema, total over 53 gigabytes of disc space for the 12 test cases. However, compressed GXL encodings of linked instances of the CppInfo schema, the smallest files recorded in this case study, total only 79 megabytes of disc space for the 12 test cases.

The results also show that the run-time costs for low-level graphs can also be prohibitive. The slowest parsing times in this case study were for compressed GXL encodings of `tu` files.

For the 12 test cases, these files took over 70 minutes to parse. The fastest parsing times in this case study were for uncompressed GXL encodings of linked instances of the CppInfo schema. For the 12 test cases, these files took just over 2 minutes to parse.

We presented results that show the importance of a linker for C++ reverse engineering tools, and presented the first experimental evidence which shows the significant savings that can be achieved by linking C++ translation units. Unfortunately, the smallest files recorded in this case study are still too large to be exchanged via email or newsgroups. This is important, as accessibility of results has been identified as a key hurdle to the adoption of existing infrastructures [Müller et al. 2000].

6.3 Case Study: Exchanging Middle-Level Graphs

In this section we describe the results of our second case study, in which we examine middle-level graphs from our infrastructure. In Subsection 6.3.1, we present results for exchanging GXL encoded instances of schemas at Levels II, III, and IV of our infrastructure. In Subsection 6.3.2, we extract results from GXL encoded instances of the Class Diagram, ORD, and Class Firewall schemas by applying XSLT style sheets. We discuss the results of the case study in Subsection 6.3.3.

6.3.1 Exchanging Graphs at Levels II, III, and IV

In this subsection we investigate the costs associated with exchanging instances of middle-level graphs. In particular, we investigate the storage costs of exchanging GXL encoded instances of the Class Diagram, ORD, and Class Firewall schemas. First, we illustrate GXL encoded instances of the ORD and Class Firewall schemas. We omit an instance of the Class Diagram, because it would be nearly identical to the ORD instance. Second, we measure the space costs incurred by exchanging graphs at Levels II, III, and IV.

In Source Listing 6.5, we list a prototypical GXL encoded instance of the ORD schema. We list two classes, `::A` and `::B`. In addition, we list an Inheritance edge, which indicates

```
<?xml version="1.0"?>
<!DOCTYPE gxl SYSTEM "gxl-1.0.dtd">
<gxl xmlns:xlink="http://www.w3.org/1999/xlink">
  <graph id="OrdInstance" edgemode="directed">
    <type xlink:href="ORD.gxl#ORD"/>
    <node id="c0">
      <type xlink:href="ORD.gxl#Class"/>
      <attr name="name"><string>::A</string></attr>
    </node>
    <node id="c1">
      <type xlink:href="ORD.gxl#Class"/>
      <attr name="name"><string>::B</string></attr>
    </node>
    <node id="e0"><type xlink:href="ORD.gxl#Inheritance"/></node>
    <edge from="c0" to="e0"><type xlink:href="ORD.gxl#isDest"/></edge>
    <edge from="c1" to="e0"><type xlink:href="ORD.gxl#isSrc"/></edge>
  </graph>
</gxl>
```

Source Listing 6.5: GXL encoded ORD instance. *A GXL encoded instance of the ORD schema containing two classes,* `::A` *and* `::B`*, and one* `Inheritance` *edge. The edge indicates that* `B` *inherits from* `::A`*.*

that `::B` inherits from `::A`. In this case, the Class Diagram instance would be identical, but for the references to the schema (shown as **`xlink:href`** attributes in **`type`** tags).

```
<?xml version="1.0"?>
<!DOCTYPE gxl SYSTEM "gxl-1.0.dtd">
<gxl xmlns:xlink="http://www.w3.org/1999/xlink">
  <graph id="ClassFirewallInstance" edgemode="directed">
    <type xlink:href="ClassFirewall.gxl#ClassFirewall"/>
    <node id="c0">
      <type xlink:href="ClassFirewall.gxl#Class"/>
      <attr name="name"><string>::A</string></attr>
    </node>
    <node id="c1">
      <type xlink:href="ClassFirewall.gxl#Class"/>
      <attr name="name"><string>::B</string></attr>
    </node>
    <node id="e0"><type xlink:href="ClassFirewall.gxl#Edge"/></node>
    <edge from="c0" to="e0"><type xlink:href="ClassFirewall.gxl#isCUT"/></edge>
    <edge from="c1" to="e0"><type xlink:href="ClassFirewall.gxl#isRetested"/></edge>
  </graph>
</gxl>
```

Source Listing 6.6: GXL encoded Class Firewall instance. *A GXL encoded instance of the Class Firewall schema containing two classes,* `::A` *and* `::B`*, and one edge that indicates that if* `::A` *has changed and must be tested, then* `::B` *must be retested as well.*

In Source Listing 6.6, we list a prototypical GXL encoded instance of the Class Firewall schema. We again list two classes, `::A` and `::B`. We also list one Edge edge, which indicates that if `::A` has changed and must be tested, then `::B` must be retested as well. This edge

Test Case	.cd.gxl[.gz]		.ord.gxl[.gz]		.cfw.gxl[.gz]	
AvP	4 207	227	5 845	301	2 735	140
CppUnit	183	10	186	10	76	8
Doxygen	4 530	245	4 309	217	2 038	100
FluxBox	1 297	71	899	49	400	24
FOX	2 922	158	582	28	953	52
HippoDraw	1 706	93	4 016	200	1 065	52
Jikes	1 345	73	4 561	221	1 041	52
Keystone	1 066	58	3 246	156	813	40
Licq	908	49	1 366	68	264	16
Pixie	1 301	71	1 988	101	693	36
Scintilla	399	22	218	12	68	4
Scribus	1 329	72	1 371	69	320	20

Table 6.8: Levels II, III, and IV: Size on disk (kB). *The size on disk, in kilobytes, for class diagrams, ORDs, and class firewalls that represent the test cases.*

results from the Inheritance edge in the ORD instance.

In Table 6.8, we list the sizes on disk, in megabytes, for class diagrams, ORDs, and class firewalls that represent the test cases. In column 1, we list the test cases. In columns 2 and 3, we list the total size of the uncompressed and compressed GXL encoded, instances of the Class Diagram schema, respectively. In columns 4 and 5, we list the total sizes of the uncompressed and compressed GXL encoded, instances of the ORD schema, respectively. In columns 6 and 7, we list the total sizes of the uncompressed and compressed GXL encoded, instances of the Class Firewall schema, respectively.

In columns 2, 4, and 6, we list the size in kilobytes[2] for compressed GXL encoded instances of the Class Diagram, ORD, and Class Firewall schemas, respectively. The average number of kilobytes for the compressed GXL encodings of instances of the Class Diagram, ORD, and Class Firewall schemas, are 95.75, 119.33, and 45.33, with standard deviances of 75.13, 96.76, and 39.61, respectively. Neither the contents, nor the sizes of these instances are directly comparable. However, the results show that none of the compressed GXL encodings for the 12 test cases is larger than 301 kilobytes, and that 25 of the 36 compressed files are no more than 100 kilobytes in size.

[2]This table uses kilobytes. The similar tables in Section 6.2 use megabytes.

6.3.2 Transforming GXL Graphs with XSLT

In this subsection we apply XSLT style sheets to the GXL instances of the middle-level graphs. In particular, we investigate the run-time costs of the transformations, and present the results for instances of the Class Diagram, ORD, and Class Firewall schemas. First, we illustrate a representative XSLT style sheet for summarizing GXL instances, in this case, instances of the ORD schema. Second, we apply XSLT style sheets to the instances of each of the three schemas, and summarize the results. We used *xsltproc* [xsltproc Project 2005] to apply the style sheets to the GXL graphs.

In Source Listing 6.7, we list an XSLT style sheet for summarizing the information in a GXL encoded instance of the ORD schema. As we noted in the introduction to this chapter, writing such a style sheet requires knowledge of only the schema, and not any particular instance. We list nine variables that contain the sets of instances of classes, edges, association edges, composition edges, dependency edges, inheritance edges, owned element edges, and polymorphic edges, respectively. We also list nine statements that print the sizes of the sets.

In Table 6.9, we present results from applying the XSLT style sheet PrintCdSummary.xslt to GXL encoded instances of the Class Diagram schema that represent each of the 12 test cases. In particular, we list the run-time costs of applying the style sheet, and summaries of the contents. In column 2, we show that *xsltproc* took less than one second to apply the style sheet to each of the test cases. In column 3, we list the total number of classes found in each instance; this class count includes all instances of the CppInfo schema classes ClassNonTemplate, ClassTemplate, and ClassTemplateInstantiation. In columns 4 through 8, we list the number of each individual edge type from the schema. Finally, in column 9, we list the total number of edges for each test case. On average, Class Diagram instances for our test cases contain hundreds of classes, and thousands of edges. Dependency edges are most common.

In Table 6.10, we present results from applying the XSLT style sheet PrintOrdSummary.xslt to GXL encoded instances of the ORD schema that represent each of the 12 test

```
<xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                           xmlns:xlink="http://www.w3.org/1999/xlink">
   <xsl:output method="text" indent="no" encoding="ISO-8859-1"/>
   <xsl:strip-space elements="*"/>
   <xsl:template match="/gxl/graph">
      <xsl:variable name="nodes"
                    select="node[type/@xlink:href='ORD.gxl#Class']"/>
      <xsl:variable name="edges"
                    select="node[type/@xlink:href!='ORD.gxl#Class']"/>
      <xsl:variable name="association"
                    select="node[type/@xlink:href='ORD.gxl#Association']"/>
      <xsl:variable name="composition"
                    select="node[type/@xlink:href='ORD.gxl#Composition']"/>
      <xsl:variable name="dependency"
                    select="node[type/@xlink:href='ORD.gxl#Dependency']"/>
      <xsl:variable name="inheritance"
                    select="node[type/@xlink:href='ORD.gxl#Inheritance']"/>
      <xsl:variable name="ownedElement"
                    select="node[type/@xlink:href='ORD.gxl#OwnedElement']"/>
      <xsl:variable name="polymorphic"
                    select="node[type/@xlink:href='ORD.gxl#Polymorphic']"/>
      <xsl:text>Nodes:          </xsl:text>
         <xsl:value-of select="count($nodes)"/>
      <xsl:text>&nl;</xsl:text>
      <xsl:text>Edges:          </xsl:text>
         <xsl:value-of select="count($edges)"/>
      <xsl:text>&nl;</xsl:text>
      <xsl:text>&nl;</xsl:text>
      <xsl:text>Association:    </xsl:text>
         <xsl:value-of select="count($association)"/>
      <xsl:text>&nl;</xsl:text>
      <xsl:text>Composition:    </xsl:text>
         <xsl:value-of select="count($composition)"/>
      <xsl:text>&nl;</xsl:text>
      <xsl:text>Dependency:     </xsl:text>
         <xsl:value-of select="count($dependency)"/>
      <xsl:text>&nl;</xsl:text>
      <xsl:text>Inheritance:    </xsl:text>
         <xsl:value-of select="count($inheritance)"/>
      <xsl:text>&nl;</xsl:text>
      <xsl:text>OwnedElement: </xsl:text>
         <xsl:value-of select="count($ownedElement)"/>
      <xsl:text>&nl;</xsl:text>
      <xsl:text>Polymorphic:    </xsl:text>
         <xsl:value-of select="count($polymorphic)"/>
      <xsl:text>&nl;</xsl:text>
   </xsl:template>
</xsl:transform>
```

Source Listing 6.7: XSLT for summarizing ORD instances. *The XSLT style sheet we used to generate the results listed in Table 6.10. We used similar style sheets to generate the results listed in Tables 6.9 and 6.11.*

Test Case	Time (s)	Classes	Association	Composition	Dependency	Inheritance	OwnedElement	Total Edges
AvP	0.66	2 099	1 353	388	6 128	371	523	8 763
CppUnit	0.02	59	27	3	349	28	6	413
Doxygen	0.57	752	406	577	6 372	492	33	7 880
FluxBox	0.15	318	163	349	1 603	233	43	2 391
FOX	0.53	500	387	352	6 311	224	203	7 477
HippoDraw	0.24	272	379	27	3 289	195	1	3 891
Jikes	0.38	433	749	150	4 645	180	55	5 779
Keystone	0.15	163	173	22	2 120	111	4	2 430
Licq	0.09	224	32	17	1 249	161	1	1 460
Pixie	0.19	309	405	30	2 296	146	50	2 927
Scintilla	0.04	89	52	79	2 198	14	1	2 813
Scribus	0.17	243	1 154	33	1 568	17	25	2 797

Table 6.9: Class Diagram sizes for the test suite. *The number of classes and edges, by type, in the 12 instances of the Class Diagram schema constructed for the applications and libraries in our test suite.*

Test Case	Time (s)	Classes	Association	Composition	Dependency	Inheritance	OwnedElement	Polymorphic	Total Edges
AvP	3.33	2 082	1 346	381	6 075	367	381	15 872	24 422
CppUnit	0.06	56	27	3	349	26	6	409	820
Doxygen	2.36	724	390	575	6 267	475	31	11 144	18 882
FluxBox	0.37	307	161	346	1 600	226	40	1 470	3 843
FOX	9.76	499	387	352	6 311	223	203	27 716	35 192
HippoDraw	2.10	271	379	27	3 289	195	1	14 043	17 934
Jikes	2.50	427	748	147	4 640	179	53	14 533	20 300
Keystone	1.78	162	173	22	2120	111	4	12 185	14 615
Licq	0.60	224	32	17	1 249	161	1	4 613	6 073
Pixie	0.91	299	398	30	2 271	142	45	5 938	8 824
Scintilla	0.24	89	52	79	2 198	14	1	469	2 813
Scribus	0.57	243	1 154	33	1 568	17	25	3 293	6 090

Table 6.10: ORD sizes for the test suite. *The number of classes and edges, by type, in the 12 instances of the ORD schema constructed for the applications and libraries in our test suite.*

Test Case	Time (s)	Classes	Edges	Min	Max	Avg
AvP	2.25	2 082	9 695	1	724	182.67
CppUnit	0.25	56	275	1	40	21.66
Doxygen	3.12	724	7 888	1	623	369.34
FluxBox	0.66	307	1 436	1	216	154.76
FOX	6.00	499	3 636	1	231	41.79
HippoDraw	0.83	271	4 200	1	210	116.07
Jikes	1.52	427	3 994	1	330	297.95
Keystone	3.33	162	3 242	1	140	87.89
Licq	0.50	224	959	1	172	26.90
Pixie	0.76	299	2 665	1	162	83.43
Scintilla	0.50	89	219	1	38	21.53
Scribus	0.62	243	1 175	1	107	64.40

Table 6.11: Class Firewall sizes for the test suite. *The numbers of classes and edges in the 12 instances of the Class Firewall schema. In addition, the minimum, maximum, and average class firewall sizes for each of the instances. Class firewall sizes are expressed as number of classes.*

cases. In particular, we list the run-time costs of applying the style sheet, and summaries of the contents. In column 2, we show that, for half of the test cases, *xsltproc* took less than one second to apply the style sheet; the maximum running time was 9.76 seconds for *FOX*. In column 3, we list the total number of classes found in each instance. This class count includes all instances of the CppInfo schema classes ClassNonTemplate and ClassTemplateInstantiation. In columns 4 through 9, we list the number of each individual edge type from the schema. Finally, in column 10, we list the total number of edges for each instance. On average, ORD instances for our test cases contain hundreds of classes, and tens of thousands of edges. Polymorphic edges are, by far, the most common.

In Table 6.11, we present results from applying the XSLT style sheet PrintCfwSummary.xslt to GXL encoded instances of the Class Firewall schema that represent each of the 12 test cases. In particular, we list the run-time costs of applying the style sheet, and summaries of the contents. In column 2, we show that, for half of the test cases, *xsltproc* took less than one second to apply the style sheet; the maximum running time was 6.00 seconds for *FOX*. In column 3, we list the total number of classes found in each instance. These classes are the same set of classes found in the corresponding ORD instance. In column 4, we list the total number of edges for each instance. On average, Class Firewall instances for our test cases contain hundreds of classes, and thousands of edges.

In columns 5 and 6, we list the minimum and maximum number of classes, respectively, found in the class firewall for any class from the particular test case. For each of the 12 test cases the minimum class firewall size is one (1). The maximum class firewall size is as small as 38 classes in *Scintilla*, and as large as 724 classes in *AvP*. The average class firewall size for all 12 test cases is 122 classes, with a standard deviation of 112.

6.3.3 Discussion

The results for exchanging middle-level graphs show, for both storage and run-time costs, savings of at least one order of magnitude when compared to the results for exchanging low-level graphs. Thus, the results indicate significant savings in the costs of exchange for applications that do not require full low-level information about a program. For example, no compressed GXL encoding of a middle-level graph is greater than 301 kilobytes for any of the 12 test cases. In addition, it took no more than 9.76 seconds to apply, using *xsltproc*, a style sheet that summarizes the contents of the given graph.

An application that builds a class firewall can take advantage of the savings that we highlight in this case study by taking ORD instances, rather than ASG or API instances, as input. This is the technique that we used to create GXL encoded instances of the Class Firewall schema for this case study. Other applications of these savings are described in Chapter 4.

We demonstrate the use of XSLT to extract information from GXL encoded instances of three different schemas. We show that this process is efficient, and present experimental results for the 12 test cases in our test suite. All GXL files that we created for this case study are available in our SourceForge.net repository, and are available for use by practictioners and other researchers.