Visual Support for Porting Large Code Bases

Bertjan Broeksema
KDAB Berlin, Germany
Email: bertjan.broeksema@fr.ibm.com[1]

Alexandru Telea
University of Groningen, the Netherlands
Email: a.c.telea@rug.nl

***Abstract*—We present a tool that helps C/C++ developers to estimate the effort and automate software porting. Our tool supports project leaders in planning a porting project by showing where a project must be changed, how many changes are needed, what kinds of changes are needed, and how these interact with the code. For developers, we provide an overview of where a given file must be changed, the structure of that file, and close interaction with typical code editors. To this end, we integrate code querying, program transformation, and software visualization techniques. We illustrate our solution with use-cases on real-world code bases.**

I. Introduction

A common problem of adapting large software systems to changing dependencies, *e.g.* libraries, is the sheer number of code changes required. For such *porting* activities, we need to realistically estimate the effort and automate changes to reduce slow, cumbersome, and error prone manual work.

We present a KDevelop extension [15] that assists porting C/C++ code bases to newer versions of their dependencies. A project-wide view shows the required changes with hints about the difficulty of each change. A file-level view shows where in a file which kind of changes are needed, and how these interact with the current code context where they are to be done. For this, we find the code fragments prone to being modified during porting by a generic, lightweight, query mechanism executed on the code base's abstract syntax graph. Next, we express automatic porting activities as source-code-rewriting rules for the queried fragments. Finally, we use several views to show the amount, type, and location of porting effort and how porting activities depend on their code context.

Section II presents related work in visual code refactoring and porting tools. Section III describes the visualizations and queries proposed in our tool: the effort estimation view, the rewrite impact view, and the impact distribution view. Section IV shows the application of our tool for porting a real-world large C++ code base. Section V discusses our techniques. Finally, Section VI concludes the paper and outlines future work directions.

II. Related work

Related work covers visualization tools and techniques for program comprehension at code level, as follows.

First, *static analysis* tools extract facts on the code to port, *e.g.* abstract syntax trees (ASTs) or annotated syntax graphs (ASGs). Program transformations use these facts to (semi)automatically rewrite code via rewrite rules. Efficient and scalable C++ analyzers include Columbus/CAN [10], EDG [8], Elsa [18], Clang [7], Eclipse's CDT [22], PUMA [1], and SolidFX [24]. C++ program transformation tools include ASF+SDF [26], Stratego [27], Transformers [2], and DMS [3]. The features of these tools vary widely, often in terms of subtle (but crucial) details, such as C/C++ dialect or template support, integration with a preprocessor, completeness and correctness of the produced ASG, range of supported transforms, and APIs for third-party tool integration. A comparison of C/C++ static analyzers is given in [4], [6]. IDEs like Visual Studio, Eclipse, KDevelop [15], and QtCreator [20] include lightweight analyzers, good for code completion and cross-references, but which cannot perform program rewrites. Static analysis also delivers code quality metrics useful to assess the porting effort [16].

Getting insight into a set of planned code rewrites is as important as doing the rewrites themselves. *Program visualization* offers several solutions. Code-level visualizations, pioneered by Eick *et al.* [9], show large code amounts with table-lens techniques [21]. Colors encode attributes such as code type and code faults [12], evolution metrics [28], or query results [24]. Program structure, dependencies, and metrics, encoded as attributed compound graphs, can be visualized using edge bundling techniques [11] or matrix plots [25], [29]. Code-level and structure-level visualizations serve complementary understanding goals by focusing on different abstraction levels.

III. Visualization tool overview

We aim to support the entire process of code porting: definition of a set of code transformations or rewrites, assessing their impact on a given code base, applying the rewrites, and assessing their effects. For this, we need insight on the impact of rewrites at several levels, such as *project level* (see the rewrites' distribution of across an entire project *e.g.* hundreds of files); *file level* (see the effect of several rewrites on a given file); and *source level* (see the effect of typically one rewrite at the level of individual code lines).

There are several tasks to support during a code porting:

Effort estimation: For a code base and set of rewrites, how are the rewrites spread over its files? This gives a good estimate of the porting effort *before* actually doing it. Code porting by rewrite engines is rarely fully automatic. Developers need to review the performed rewrites to ensure that these are indeed correct and desirable, *e.g.* do not change the code to alter program semantics or given coding styles. Few rewrites or rewrites grouped in a few files indicate minor, localized, changes which arguably take less effort to understand, execute, or review than many rewrites spanning the whole code base.

Rewrite impact: When the same code fragment is affected by several rewrites, their order may be important. Even when the

[1]B. Broeksema is now with IBM Center for Advanced Studies, France

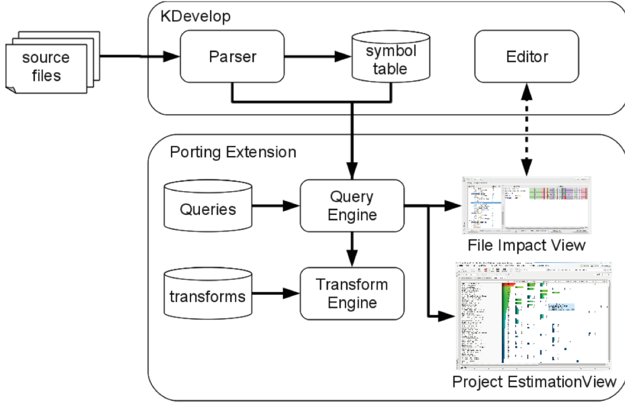

Fig. 1. Overview of our code porting framework

semantic effect is order independent, different orders may lead to different code layouts, some of which are less readable than others. Depending on the rewrite engine and actual rewrites used, some rewrites may be conflictual; getting an overview of code fragments affected by such rewrites is useful.

To support the above tasks, we have developed a porting engine, or extension, for the KDevelop IDE (Fig. 1). We use KDevelop's C++ static analyzer to extract ASG data from a given code base. Second, we created a query engine which finds code fragments (and their ASGs) that match user-specified patterns which have to be rewritten. These patterns are next transformed in the ASG based on user-specified rewrite rules; this is the actual porting. Third, we created a set of views which address the effort estimation and rewrite impact understanding tasks at project, file, and source code level. The query, rewrite, and visualization engines are integrated in KDevelop as linked views, which lets one query, rewrite, and visualize code in an integrated way (see Fig. 2 for an overview).

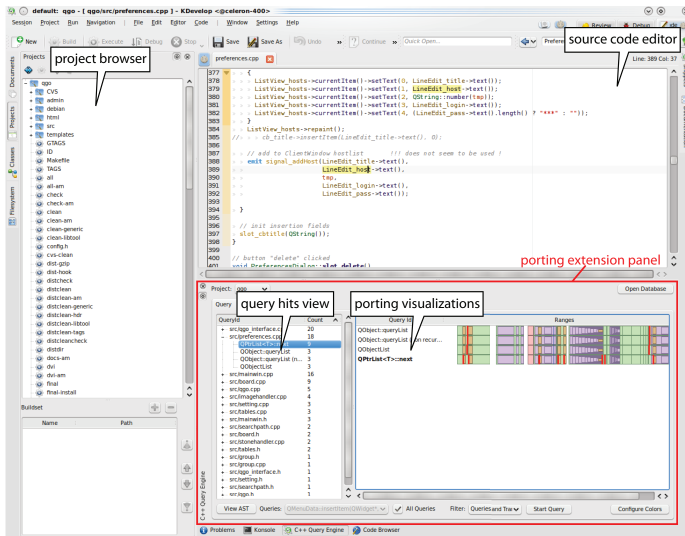

Fig. 2. KDevelop IDE with integrated porting extension

We describe the components of our porting extension.

A. Querying and transformation

To support automated porting, we must first describe which code constructs are affected by porting. Examples are usage of a given API (set of functions or classes); inclusion of certain headers; and usage of certain language features *e.g.* constructors, default arguments, or templates which is specific to a given API version. We describe constructs of interest as results of *queries*

$$Q(A \times S_Q) \rightarrow \{H\star\}, \quad Q(a \in A, s_Q \in S_Q) = \{h_i \in a\} \tag{1}$$

A query takes an AST $a \in A$ and a query pattern $s_Q \in S_Q$ and produces a set of *hits* $h = \{h_i\} \in a$, *i.e.* AST nodes which match the query pattern. Hits also have location (token, line, column) data provided by the KDevelop parser. Our query language S_Q is largely similar with the one of the EFES and SolidFX C++ static analyzers [4], [24]. We also find hits by matching patterns with actual AST fragments with a visitor design. A full specification of our query language is given at [6]. Querying uses KDevelop's *DUchain* (definition-uses chain) system and it correctly finds all uses of a symbol, *e.g.* function, type or variable, across translation unit boundaries. This is essential for the next step: program-wide code rewrites.

A query outputs code fragments which we may want to change via automatic program rewriting. Two main techniques exist here. *Procedural rewriting* changes selected AST fragments (our hits h_i) based on given rules [2], [17]. However generic, this approach has some problems. Data not contained in an AST, such as code layout, comments, or macros, is hard to maintain. This may generate code which is correct but otherwise illegible, and may loose lexical-level information. DMS alleviates this by intermixing C++ preprocessing and parsing at the expense of a more complex implementation [3]. *Source-to-source rewrite* rules, in contrast, work directly at source-level, *i.e.* on the actual tokens. This method is far easier to implement than the one in DMS, and is sufficient for the less general transforms needed for porting, *e.g.* identifier renaming, changes of function signatures and call arguments, and changes of scope qualifiers for symbols. We define a transform as:

$$T(S_Q \times H \times S_T) \rightarrow C, \quad T(s_Q \in S_Q, h \in H, s_T \in S_T) = c \in C \tag{2}$$

A transform T takes a hit $h = Q(a, s_Q)$ from applying a query s_Q on an input program a (Eqn. 1), and a set of rewrite rules S_T, and produces a source-level change c of the code in h. Rewrite rules can refer to specific syntactic parts of the query result, as captured by the query pattern s_Q. In this way, we can rewrite conditionally on the *values* of specific elements, *e.g.* only rename a symbol if it occurs in a scope with a given name. A rewrite rule performs insertion and replacement actions on the code it acts. Like conditions, insertions and replacements can act on specific code parts as described by the rewrite specification s_T. For instance, we can erase the last argument of a function call, or qualify a symbol with desired scope names. A full description of the transform language S_T is given in [6].

Source-to-source transforms using code rewrite rules have several advantages. First, rewrite *actions* can ignore KDevelop's internal C++ grammar representation. We have found this to be highly desirable for developers who do not want to get deeply involved with such issues. Secondly, the rewrite-engine

stays relatively simple, and still can keep source code layout, comments, and macros largely similar between original and transformed code. The key to this is that rewrite rules specify code to be changed in terms of *syntax*, but do the actual changes (insertion on replacement) in terms of *ranges* (start-end locations) in the code. This fully preserves layout, preprocessing macros, and comments outside changed code. Since KDevelop's ASG captures semantic data, relatively complex changes such as replacing a symbol in all scopes where it is used, or changing all instances of a given C++ template, work as expected.

However, our solution also has some limitations. Our rewrite rules are local, *i.e.* can only modify code within their input hit range. This excludes transforms such as rewriting code around a function call when the function's return type is changed. Secondly, if different queries yield overlapping hits, we cannot apply a set of transforms in a single step, but need to perform pairs of query-and-transform steps one at a time.

Given the above, user inspection of the effects of a given query-and-transform set is important. We next present a set of visualizations which address this concern and, at a higher level, support developers in planning and executing porting tasks.

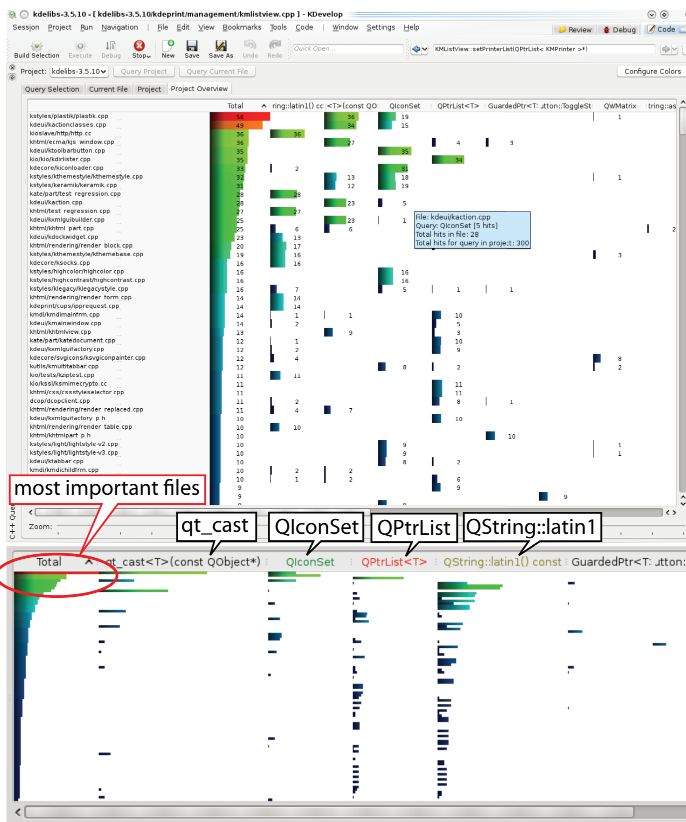

Fig. 3. Effort estimation view, sorted on total hits per file

B. Effort estimation view

Our first view addresses the task of estimating the overall effort of porting the *kdelibs 3.5.10* code base. Our code base has to be ported from version 3 of the well-known user interface C++ Qt library (Qt3) to version 4 (Qt4). We use a table-lens approach (see Fig. 3). The query set (15 queries shown here out of a few tens in total) captures code patterns involved in this port, following [19]. For each of the 550 files in the code base, a table row shows the number of hits for each query in the query set. The first column shows the sum of all hits for the respective file. The table can be zoomed out, reducing rows to colored pixel bars, and sorted on any column. The bottom image shows files sorted by total number of query hits. We see here that most porting effort is located in about 5 to 10% of the entire set of files. The most important queries, thus main types of changes to be done, are *qt_cast* (Qt typecast macro), *QIconSet* (construct QIconSet objects), *QPtrList<T>* (templated object pointer containers), and *QString::latin1* (string localization), *i.e.* table columns 2 to 5, since these have most cells with large values. To further support effort estimation, developers may define, for each query, a porting difficulty level, which reflects how well the rewrite engine handles that specific construct (Sec. III-A). Column headers are colored to reflect this difficulty with a green-red (easy-hard) colormap. In Fig. 3, we see for instance that *QString::latin1* is frequent, but can be easily ported; however, *QPtrList<T>* also affects many files, but has a high porting difficulty. Overall, the porting view assist porting decisions *e.g.* trigger the development of new rewrite rules; allocate specific developers to specific project parts; and choose to do some porting activities by hand and automate others.

C. Rewrite impact view

As explained in Sec. III-A, code rewriting is rarely fully automated. Apart from our rewrite engine limitations, we have seen that developers first want to see what such an engine would change in their code before firing it off, especially for code bases having many rewrite locations (hits). Typical questions are: which code fragments are affected by a given rewrite rule, or by more rules; and how do rewrites spread over a file (*e.g.* are they condensed in specific parts like the leading include or declaration sections, or do they affect the whole file).

The *rewrite impact view* addresses these questions (Fig. 4). A tree browser in the left panel shows all files in the code base, and the specific queries and query hits for each file. When we select a file in this view, the right panel shows the query hits in that file. In this panel, the x axis maps to the file extent (left=first line, right=last line). Each horizontal bar shows hits for a specific query; hits are red blocks. The bar itself is a condensed view of the code in the queried file. To render this, we traverse the file's AST in depth-first order and render each node as a bar block. The block's x position and width reflect the node's size (in lines of code, LOC) and location in the file. The block's height reflects the nesting level: deeper nodes are drawn as blocks inside their parents' blocks. Block colors show AST node types, *e.g.* brown for loops (for, do, while), purple for control statements (if, switch), cyan for C-style functions, green for public methods, orange for protected methods, and red for private methods. This colormap covers only a small subset of all C++ AST node types *i.e.* the constructs of interest during porting. Node types not in the colormap and nodes whose blocks are smaller than a few pixels in either dimension are not drawn, so that the view stays uncluttered. The red hit blocks are always drawn at full height; their width shows the hit range. Hit block widths are limited to a few pixels so hits always stay visible.

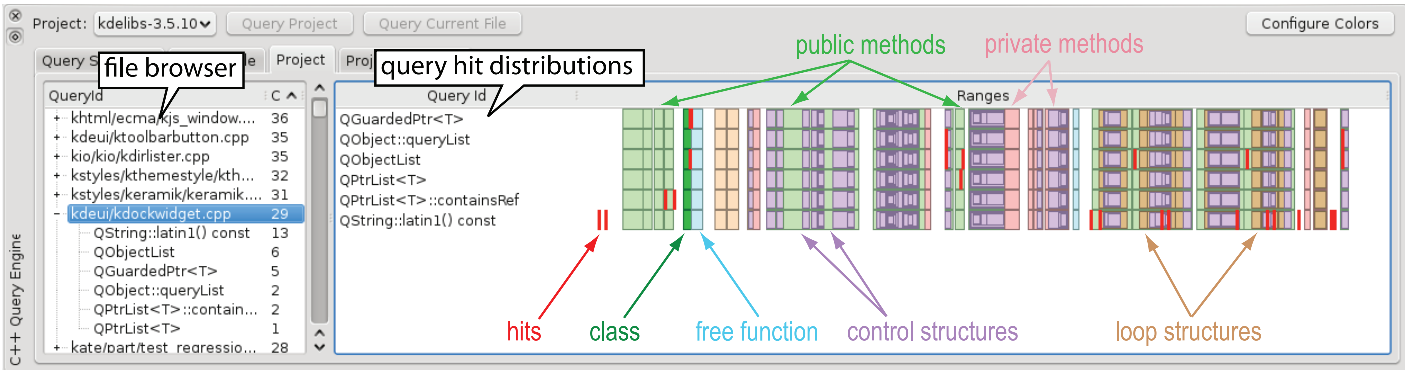

Fig. 4. Rewrite impact view with syntax structure of a selected files overlaid by query hits (see Secs.III-C and IV-B)

The rewrite view can be zoomed on the x axis like a table lens. Zooming in makes the screen-size of deeper nested nodes larger, so these become visible. Zooming out emphasizes large, global-scope constructs, *e.g.* function definitions and class declarations.

The rewrite impact view outlines query hit distribution at file level. Vertical red bars show code ranges with overlapping hits, thus areas of potential automatic rewrite problems. The spread of red bars in the file indicate where rewrites will occur in the file, *e.g.* in the preamble or main code range, and also with respect to code constructs, *e.g.* within function or class declarations. This view is linked to KDevelop's editor by brushing and selection, so developers can examine in detail the potential effects of a given rewrite before actually doing it.

IV. Applications

We now present three use-cases for our visual porting support. As input, we take the *kdelibs* C++ code base (750 KLOC, 550 files) and the Qt3 to Qt4 porting scenario along the official porting guidelines [19]. Porting code from Qt3 to Qt4 is mainly related to rewriting code which uses Qt3 API calls or data types to their Qt4 equivalents. This involves several syntactic and semantic changes related to the evolution of the Qt toolkit API.

A. Estimating the effort

When starting a porting job, developers want a quick assessment of the challenge at hand, *i.e.* the porting effort and type of work involved (automatic vs manual rewriting). The effort estimation view (Fig. 3), sorted by total aggregated query hits per file, supports this. For *kdelibs*, this view shows that the main effort is condensed in 5 to 10% of the project files; most rewrite actions can be done automatically; but there are a few such actions, like the rewriting of *QPtrList*$<T>$ template class uses, which the current rewrite engine cannot handle (see Sec. III-B). Hits of a given query in a given file can be examined in detail by clicking on their cells in the table-lens view in Fig. 3, which selects their code is KDevelop's editor.

We should stress that porting effort *estimation*, based on query hits, and actual porting *execution*, based on automatic and/or manual rewriting, are separate activities. The effort estimation view only highlights how much and where must work be done, and if this can be automated or not. Developers are free in how they do the rewriting, *e.g.* use our own rewrite engine (when fully automatic transforms exist for specific queries), manually rewrite the code, or use third-partly rewrite engines such as, in our case, the *qt*32*qt*4 engine provided by Qt itself. The developer is not obliged to provide transforms for *all* queries pertaining to a given porting task.

B. Performing the porting

After the developer has assessed that the porting effort for a code base is acceptable, actual porting starts. Here, one would like to use the rewrite engine as much as possible. Since, as already explained, this may not work correctly in all cases, a good strategy is to examine the code file by file, or query by query, and assess when automatic rewriting is safe. For this, we use the rewrite impact view (Fig. 4).

Fig. 5. Potential rewrite inspecting (top). After the rewriting (bottom)

Several points can be made here (see Fig. 5). The selected file contains many public methods, as shown by the many small light green blocks. Apart from these, we see a few private methods halfway the file (pink blocks) and a few protected methods in the left-middle area (orange blocks). The red hits

in the white areas near the beginning of the file (left region) show code to be ported as well, located in very small functions, whose screen size is under a few pixels in this view. We also see some larger functions containing consecutive and nested control structures (purple blocks) and loops (brown blocks). Finally, we see a class definition (dark green block at the beginning).

Next, the developer wants to examine change locations in more detail. This serves to understand why certain hits occur; for instance, hits of a certain query in a file may show constructs which one did not expect to be present. This occurs *e.g.* for code bases maintained by many developers, such as in open-source projects. Another use-case is examining overlapping hits, *i.e.* vertical red bars; these are potential automatic rewriting problems. For details, the user clicks in an area of interest in the rewrite impact view. This opens a KDevelop editor for the respective file and places the edit cursor at the location corresponding to the x position clicked in the bar. Clicking on a red hit block selects the hit's code range in the editor (see Fig. 5 top). The location selected in the rewrite impact view, matching the cursor position in the editor, is shown in this view as a thin blue marker. The marker is updated when the user clicks on the color bar and when moving the cursor in the editor, so the editor and rewrite impact view are linked in both ways.

In our case (Fig. 5 top), the hit is a Qt3 *QGuardedPtr<T>* construct, to be ported to its Qt4 syntax (*QPointer<T>*). Note that this query correctly handles C++ template instantiations, in our case the instance being *QGuardedPtr<KHTMLPart>*. Now the user can decide how to do the rewrite. Here, he uses the rewrite engine, since the construct occurs in a simple context, and, from the rewrite rule details, one knowns that this rewrite has no side effects. A right-button click menu on the hit (not shown in the image) does the rewriting. Figure 5 bottom shows the rewriting effect: The *QGuardedPtr<T>* hits have vanished from the list of query hits in the rewrite impact view, which has now three queries (colored bars) as compared to four before rewriting. The code in the editor is updated automatically, since the rewrite affects the underlying source file. Manual rewriting works conversely: the user changes the code in the editor and the impact view is updated.

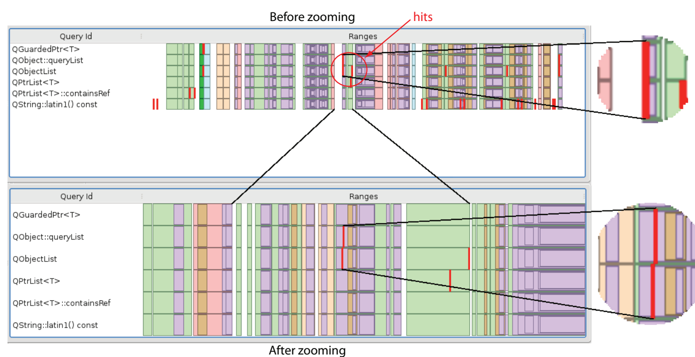

Fig. 6. Zooming the rewrite impact view to analize query hits

We next show the zooming the rewrite impact view. This helps deciding whether certain hits which seem to overlap in the zoomed-out view truly refer to the same code range or not. The mouse wheel zooms the view around the mouse position. Figure 6 shows how this works. Before zooming, we see two groups of two hits each, which seem to overlap (Fig. 6 top). After zooming in this area (Fig. 6 bottom), we see that the first two hits are closely spaced, but not overlapping, inside one control structure (purple block). The two other hits, inside a public method (green block), fall further apart. As no hits overlap, they can be ported in one step using the rewrite engine.

C. Complexity assessment for affected code

The decision to rewrite automatically or manually, rewrite ordering, and the overall rewrite difficulty depend on more than the query hit count, hit locations, and hit overlaps. Other factors include the overall *context* of a code fragment, *e.g.* location, surrounding code constructs, and related comments. To support more kinds of reasoning about the impact of a rewrite on a given file, we added additional metrics in the rewrite impact view. Different color schemes serve different analyses. One example which proved useful was to compare changes in public method declarations to changes in protected or private declarations. The hypothesis is that changes in public declarations have a higher impact on a code base than changes on protected or private declarations. Hence, a rewrite of a public declaration should be done with more care (if done manually) than one of protected or private declarations. Similarly, changes in the implementation of a method are more localized than changes in the (public) interface of a class. However, in this case, the complexity of the code around the change is important. For example, rewrites in deeply nested control and loop structures or in C-style casts potentially make the code more unreadable than rewrites in simpler code like assignment sequences, since the former are already more complex than the latter even before rewriting [23]. Hence, rewrites on complex structures should be done with greater care.

To support such analyses, we use a colormap which depicts code complexity with respect to rewriting. All top-level structures such as class declarations and function bodies get the same light gray tint to show an overview of global structure. Loops have different tints of green (depending on the loop type *e.g. for*, *do-while*). Control structures have different purple tints (*if*, *switch*, etc). C-style casts are light blue.

Figure 7 shows this colormap for three files of the *kdelib* code base. The first file (A) is a header with no implementation (*e.g.* inline methods) and thus has low complexity, as shown by the gray-tint bar. For the next two files (B,C), we show the whole file and a zoom-in of a complex part thereof. We see that these files contain different kinds of deeply nested loops, nested control structures, and several C-style casts. In file B, hits show *QString::latin1()* function calls, which returns a Latin-1 encoding of a string object. In file B, we see a recurring pattern: a *do-while* loop (light green) which ends with several C-casts (nested cyan block), and has a hit in the last statement (red line) *i.e.* in the loop control expressions containing *QString::latin1()* calls. Also, we see that all hits are clustered within a single large block roughly halfway the file, where we zoomed in. Here is a method definition. For file C, we display hits for six different queries. The first five queries have relatively few, clustered, hits as shown by the red bar locations (Fig. 7, file C, unzoomed

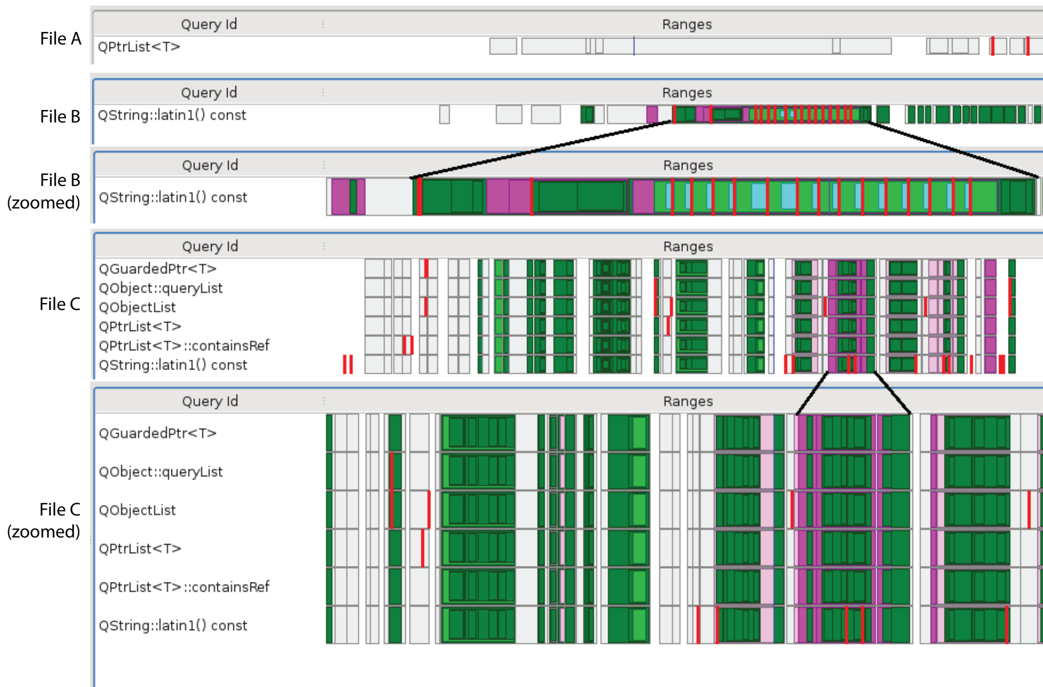

Fig. 7. Rewrite impact view colored to emphasize code complexity with respect to rewriting (see Sec. IV-C)

view). The last hit, again for the *QString::latin1()* query, has more hits which are spread over a larger file portion. However, few hits overlap, so porting can be done by manually rewriting these overlapping regions followed by automatic rewriting of the remaining cases. Zooming in over a small range of file C shows that most hits are outside complex structures, except the *QString::latin1()* hits (red bars over green blocks).

D. Anticipating porting effort

For code bases which depend on third-party components, porting is rarely a one-shot activity. When the interface of such a component changes, the code base is likely to need rewriting. Such porting efforts can be anticipated by checking the code base's usage of so-called deprecated APIs. These are APIs which are (highly) likely to be dropped off in future releases, thus whose usage implies future porting costs. The *kdepimlibs* library, part of the KDE framework [14], which contains code for personal information management (PIM) functions, is a good example. This library is widely used in the *kdepim* component of KDE. The evolution of *kdepimlibs* featured several tens of deprecated methods. Within this library, deprecated methods are marked by special macros. We developed a query set for finding such methods which resulted in 47 function queries spread over 15 classes of *kdepimlibs*. Next, we checked how a new release of *kdepim* (to be included with KDE 4.5) uses such methods by searching this release with our query set.

Figure 8 shows the results of this analysis. Strikingly, there are only 9 uses of 6 deprecated APIs ($m1,\ldots,m6$) within *kdepim*, even though the size of the latter is over 500 KLOC. This is a good signal *i.e.* very low effort for porting *kdepim* with respect to *kdepimlibs* API deprecation. As a reference, we queried *kdepim* for usage of non-deprecated *kdepimlib* functionality, *i.e.* the use of APIs in the *Akonadi::Collection* class, which is a non-deprecated part of *kdepim*. This yielded 2043 hits in 373 files. This implies that *kdepimlib* is indeed a non-trivial dependency of *kdepim*. Further inquiries revealed a possible explanation for the low usage of *kdepimlibs* APIs within *kdepim*: the two components are largely maintained by the same team, which suggests that concerted efforts have been done to remove usage of deprecated APIs.

Fig. 8. Deprecated API usage for anticipating porting efforts (Sec. IV-D)

E. Assessing porting dependencies at system level

The effort estimation view (Fig. 3) only shows the relation of porting changes (query hits) to code at the lowest level (files). For large systems, developers may want to assess how a set of (porting) changes affects their code at higher abstraction levels, *e.g.* subsystems. For this, we use an existing dependency visualization technique in a new way. Given a query Q and a code base with a set of files F, we first compute all files $f_Q \subset F$ which contain at least one hit $h \in Q(f \in F)$. Next, we construct a compound graph G consisting of containment and dependency relations. Containment relations reflect the software structure (folders and files), readily available from KDevelop's static analysis. Dependency relations link all file nodes belonging to the same set f_Q, for all queries Q. This reflects that porting the code with respect to a query Q needs to modify all files f_Q.

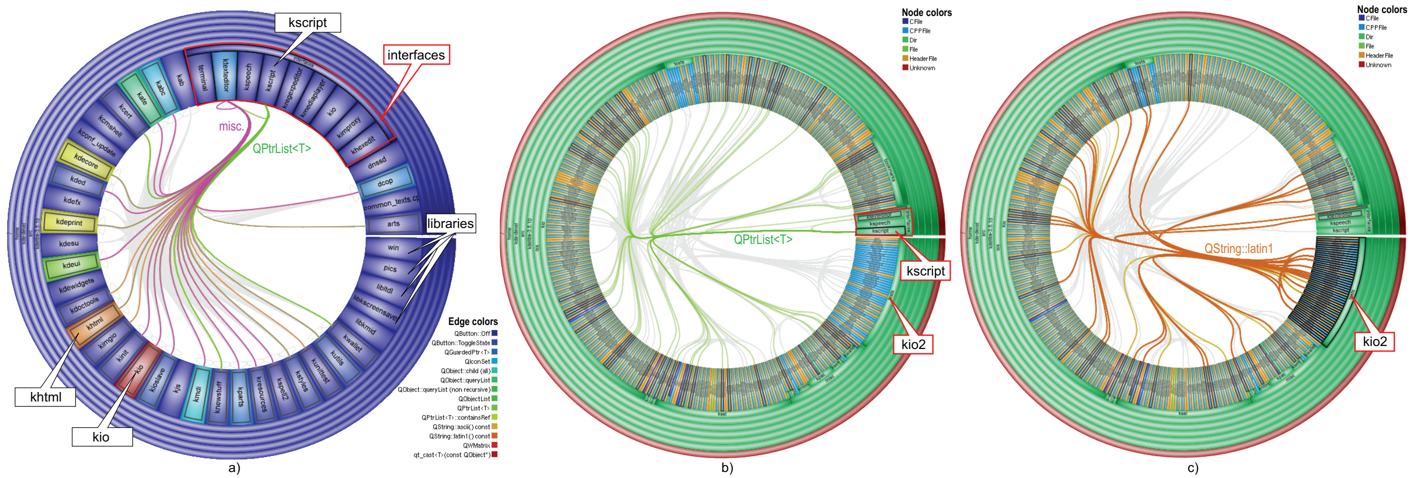

Fig. 9. Understanding porting dependencies by visualizing query hit relations between subsystems (Sec. IV-E)

To visualize *G*, we use the hierarchical edge bundling (HEB) technique of Holten *et al.* [11], adapted to encode porting attributes[1]. Figure 9 shows this for our *kdelibs* example. The left image (Fig. 9 a) shows a high-level view of the code. Nodes are folders, colored by the number of query hits with a blue-to-red rainbow colormap. Several folders stand out as containing many hits, *e.g. kio* and *khtml*. Next, we select the *interfaces* folder and only show query hits relating this folder to the rest of *kdelibs*, *i.e.* the KDE packages, colored by query type. This shows that the Qt3-to-Qt4 changes in the *interfaces* folder which affect packages involve *QPtrList<T>* constructs in the *kscript* sub-interface (green edges) and several other constructs in the *ktexteditor* sub-interface (purple edges). We next zoom in on the porting dependencies between *interfaces* and *kio*, by expanding *kio* to file level over the entire circumference of the HEB view (Fig. 9 b). We now color nodes based on their type (folders=green, C++ files=blue, headers=orange). When we select *kscript*, we see now precisely which files share *QPtrList<T>* porting dependencies with *kscript* - these are the nodes connected by green edges with the highlighed *kscript* node in Fig. 9 b, *i.e.* about 30% of all files in *kio*, spread over most of the *kio* subsystems. Porting dependencies of other types than *QPtrList<T>* are shown in gray. We see a large bundle of these in the lower-right area, and select them for further inspection (Fig. 9 c): We now see that these are *QString::latin1* porting changes (orange edges) which relate to the *kio*2 subsystem we just selected; they go to sibling subsystems in *kio* and none to *kscript*. Hence, a Qt3-to-Qt4 porting affects *kio* at *implementation* level via *QString::latin1* constructs and at *interface* level via *QPtrList<T>* constructs due to the *kscript* API.

V. Discussion

Ease of use: Our views, query engine, and rewrite engine are tightly integrated with KDevelop. For this, we reuse the open APIs of KDevelop for ASG access and GUI management. The incremental code analysis in KDevelop ensures that queries and transforms done on large code bases show their results in the views on-the-fly as these become available, which gives a smooth experience.

Queries and transforms are declaratively written in XML. Although this offers less freedom than *e.g.* using an imperative query and/or transform language, it reduces end-user effort. Typically, users start with an existing (XML-based) query or transform set, and modify these gradually to suit their needs.

Performance: The speed of KDevelop's C++ analyzer (slightly higher than compilation time) is key to the performance of our solution. For the entire *KDE* code base (3.8 MLOC), this is 36 minutes on a 1.8 GHz machine with 2 GB RAM. Although this may sound high, note that analysis is done once, and only redone for those files which are changed. In practice, this yields near-real-time response time for typical developer activities.

Generality: Our porting support solution depends on KDevelop only in terms of implementation. The query, rewriting, and visualizations techniques presented are generic and do not rely in any way on C/C++ specifics. The only requirement is the availability of a static analyzer that produces an ASG with source code locations. Hence, our solution can be readily integrated *e.g.* in QtCreator [20] (which also has an API for its internal C/C++ analyzer), KDevelop for other languages than C/C++, or Eclipse (*e.g.* via the CDT C/C++ or Recoder Java analyzers [22], [17]). The Recoder framework is particularly suited, as it offers a powerful API for code rewriting.

Validation: We have used our framework for several real-world code porting contexts: work done at KDAB, Inc. [13], a company specialized in software porting solutions and in particular Qt-based code; and code refactoring work in the KDevelop open-source project after the KDE 4.4 release. In both cases, our framework has been able to handle complex code bases of millions of LOC and help developers save valuable time during porting activities.

We also compared our solution to *qt32qt4*, the official Qt3 to Qt4 porting tool in the Qt SDK. The aim of this tool is largely similar to ours: assist developers in porting Qt3-based code to the Qt4 API. *qt32qt4* uses a lightweight C++ analyzer

[1]For HEB technical details, we refer to the paper of Holten [11]

to find and rewrite code fragments which comply with a built-in list of porting patterns. Given this, *qt32qt4* has several serious limitations, including incorrectly rewriting syntactically similar, but semantically different code (due to scoping and lookup limitations), and not rewriting certain constructs. This results in broken code that does not compile, or worse, compiles but executes with different semantics. In contrast, our solution has less limitations: although it cannot perform any type of code transformation, the rewrites it handles do not result in compiling but incorrect code. Moreover, our solution is more generic, *i.e.* it can be used for C++ rewriting beyond porting Qt-based code.

Limitations: For code analysis, we are limited by the quality of ASG information provided by KDevelop's own C++ analyzer. In particular, this analyzer does not perform a so-called *elaboration* phase on the ASG, *i.e.* the insertion of non-explicit constructs for implicit cast operator calls and constructor calls, and destructor calls for stack objects. However, such constructs need to be handled during porting to maintain code semantics. Other C++ analyzers such as Clang [7], Elsa [18] or SolidFX [24] do this and thus provide a richer ASG. Currently, we solve this problem by inserting on-the-fly checks in our queries and transforms to account for such constructs. However, this solution is not ideal, as such code should belong to the C++ analyzer proper.

Our query engine is simpler than general-purpose ASG pattern-matching engines like [24]. Specifically, we do not support querying for patterns involving all nodes present in the C/C++ grammar, but limit ourselves to the most interesting ones for (API-related) porting scenarios, *i.e.* nodes which describe the usage of an external API in terms of types, inheritance, templates, class member access, and function calls. If desired, all C/C++ AST nodes could be added to this model, resulting in a query engine very similar to [24]. Similarly, our code rewriting engine is geared towards changes which are *local* to a given query context. For porting code, this is however adequate, since typical changes here do not spread over large amounts of code.

It is tempting to consider third-party C++ analyzers to remove KDevelop's analyzer limitations. There are few open-source C++ analyzers which do preprocessing, have code location data, provide rewriting, are scalable, and cover C/C++ fully. The only suitable candidate we know is Clang [7]. However, integrating Clang in KDevelop is not trivial. Given these reasons, our solution is a good compromise between generality and development effort.

Availability: Our framework, including examples, is openly available as a KDevelop 4.0.1 extension [5].

VI. Conclusions

In this paper, we have presented a visual assistant for porting C/C++ code bases. Our solution assists users in assessing the porting effort for a given code base, determining possible conflicts (and resolutions thereof) during porting, getting an overview of the issues affecting code under porting, and doing the porting semi-automatically. We presented two new visualizations: the effort estimation view and the rewrite impact view, and used the existing HEB visualization in a new way. Examples are shown for industrial and open-source code bases.

We next consider more generic, easier to specify ways to select and transform code fragments. Also, we plan to augment our visualizations to support more complex understanding scenarios such as the assessment of ripple effects determined by a certain code change throughout an entire code base and semi-automatic what-if scenario support for assessing the impact of a given (set of) code modification(s) before these are actually executed manually or automatically.

References

[1] R. Akers, I. Baxter, M. Mehlich, B. Ellis, and K. Luecke. Reengineering C++ component models via automatic program transformation. In *Proc. WCRE*, pages 167–175, 2005.

[2] R. Anisko, V. David, and C. Vasseur. Transformers: A C++ program transformation framework. tech. rep. 0310, EIPTA/LRDE, France, 2003.

[3] I. Baxter, C. Pidgeon, and M. Mehlich. DMS: Program transformations for practical scalable software evolution. In *Proc. ICSE*, pages 234–243, 2004.

[4] F. J. A. Boerboom and A. A. M. G. Janssen. Fact extraction, querying and visualization of large C++ code bases. MSc thesis, Dept. of Comp. Sci., Eindhoven Univ. of Technology, the Netherlands, 2006.

[5] B. Broeksema. KDevelop C++ query and rewriting extension, 2010. http://www.gitorious.org/kdevcpptools/kdevcpptools.

[6] B. Broeksema. A visual tool-based approach to porting C++ code. MSc thesis, Dept. of Comp. Sci., Univ. of Groningen, the Netherlands, June 2010. http://www.cs.rug.nl/svcg/SoftVis/Refactor.

[7] Clang Team. The Clang C++ analyzer, 2011. http://clang.llvm.org.

[8] Edison Design Group. EDG C++ front-end, 2011. www.edg.com.

[9] S. Eick, J. Steffen, and E. Sumner. Seesoft - a tool for visualizing line oriented software statistics. *IEEE Trans. Softw. Eng.*, 18:957–968, 1992.

[10] R. Ferenc, A. Beszédes, M. Tarkiainen, and T. Gyimóthy. Columbus - reverse engineering tool and schema for C++. In *Proc. ICSM*, pages 172–181, 2002.

[11] D. Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE TVCG*, 12(5):741–748, 2006.

[12] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proc. ICSE*, pages 132–140, 2002.

[13] KDAB Inc. KDAB company, 2010. www.kdab.com.

[14] KDE PIM Team. KDE PIM personal information management library, 2011. http://community.kde.org/KDE_PIM.

[15] KDevelop team. KDevelop IDE for C++, 2010. www.kdevelop.org.

[16] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice*. Springer, 2005.

[17] A. Ludwig. Recoder Java static analyzer and program rewriter, 2010. http:recoder.sourceforge.net.

[18] S. McPeak. The Elsa C++ parser, 2011. www.scottmcpeak.com/elkhound/sources/elsa.

[19] Nokia, Inc. Qt 3 to Qt 4 porting guide, 2011. http://doc.qt.nokia.com/4.6/porting4.html.

[20] Qt Creator team. Qt Creator integrated development environment, 2010. qt.nokia.com/products/developer-tools.

[21] R. Rao and S. K. Card. The table lens: merging graphical and symbolic representations in an interactive focus + context visualization for tabular information. In *Proc. ACM CHI*, pages 234–242, 1994.

[22] D. Schaefer. The Eclipse C++ development toolkit, 2011. www.eclipse.org/cdt.

[23] H. Sutter and A. Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and best practices*. Addison-Wesley, 2005.

[24] A. Telea, H. Byelas, and L. Voinea. A framework for reverse engineering large C++ code bases. *Electr. Notes Theor. Comp. Sci.*, 233:143–159, 2009.

[25] F. vam Ham. Using multilevel call matrices in large software projects. In *Proc. InfoVis*, pages 227–232, 2003.

[26] M. van den Brand, J. Heering, P. Klint, and P. A. Olivier. : Compiling language definitions: the ASF+SDF compiler. *ACM Trans. Program. Lang. Syst.*, 24(4):334–368, 2002.

[27] E. Visser. Program transformation with Stratego/XT. technical report UU-CS-2004-011, Institute of ICS, Utrecht Univ., Netherlands, 2004.

[28] L. Voinea and A. Telea. CVSgrab: Mining the history of large software projects. In *Proc. EuroVis*, pages 198–206, 2006.

[29] D. Zeckzer. Visualizing software entities using a matrix layout. In *Proc. ACM SOFTVIS*, pages 210–217, 2010.