Noname manuscript No.
(will be inserted by the editor)

The Modular and Feature Toggle Architectures of Google Chrome

Md Tajmilur Rahman · Peter C. Rigby · Emad Shihab

Received: March 9, 2017/ Accepted: June 8, 2018

Abstract Software features often span multiple directories and conceptual modules making the extraction of feature architectures difficult. In this work, we extract a feature toggle architectural view and show how features span the conceptual, concrete, and reference architectures.

Feature toggles are simple conditional flags that allow developers to turn a feature on or off in a running system. They are commonly used by large web companies, including Google, Netflix and Facebook to selectively enable and disable features. Recently, libraries to help inject and manage toggles have been created for all the major programming languages.

We extract the feature toggles from the Google Chrome web browser to illustrate their use in understanding the architecture of a system. Since there is no high-level conceptual and concrete architectures for Chrome, we had to manually derive these representations from available documentation and map them into the source code. These modular representations of a modern web browser allowed us to update the 12 year old research on browser reference architectures with current technologies and browser concepts.

Mining the usages of feature toggles in the source code, we were able to map them on to the modular representation to create a feature toggle architectural view of Chrome. We are also able to show which features are contained in a module and which modules a feature spans. Throughout the paper, we show how the feature toggle view can give new perspectives into the feature architecture of a system.

Md Tajmilur Rahman
Concordia University, Montreal QC, Canada E-mail: mdt_rahm@encs.concordia.ca

Peter Rigby
Concordia University, Montreal QC, Canada E-mail: peter.rigby@concordia.ca

Emad Shihab
Concordia University, Montreal, QC, Canada E-mail: eshihab@encs.concordia.ca

Keywords Software Architecture · Architecture Extraction · Feature Architecture · Toggles, Switches, Flags

1 Introduction

Software architecture plays a key role in software maintenance and evolution. Software engineers can refer to the architecture to understand the software system before modifying or re-engineering it. Traditionally, the software architecture is derived from the existing documentation and the relation among source files, such as the call graph. For example, Bowman et al (1999) architectural extraction process involves using the conceptual architecture diagrams and relations in the directory structure to reconstruct the concrete architecture.

Since traditional architectural representations are usually at the module level, features are often difficult to represent as they can span multiple modules and are not explicitly delimited in the source code. For example, a feature as simple as logging into a system will *span* three layers requiring a user interface, a database with user names, and business logic to check that the password is correct for the user name.

Research into software product lines has focused on use case [Griss et al (1998)] and UML representation of features, however, it is difficult to project these separations of features into the code base and to keep the UML representation up-to-date [Czarnecki et al (2004); Benavides et al (2005)]. In industry, separate software product lines are often achieved by cloning code and modifying it to conform to the different feature requirements [Rubin et al (2015); Dubinsky et al (2013)]. This can lead to long-term maintenance issues as the two code bases evolve differently. It is also difficult to capture a single architectural diagram of the cloned code [Dubinsky et al (2013)].

Recent developments at large web companies, such as Google and Facebook, have seen the need to isolate each feature so that the feature can be quickly turned off during botched releases and for feature comparison testing, such as A/B testing. To achieve this flexibility, many companies use feature toggles [Dixon et al (2011); Fitzgerald and Stol (2014)]. The use of feature toggles is becoming generally accessible to software engineers with the development of toggle libraries that ease injection and management of toggles for each programming language [FeatureFlags (2018)].[1] The "promises and perils" of feature toggles were recently studied in a practitioner-focused work by Rahman et al. [Rahman et al (2016)]. The goal of the current work is to understand how feature toggles affect the architecture of a system.

[1] For example, Java has 5 competing toggle libraries `http://featureflags.io/java-feature-flags/` accessed May 20, 2018

1.1 What is a feature toggle?

Feature toggles are conceptually simple, they combine flags with conditionals to denote portions of the code that are related to a feature. Unlike compile time toggles (compile-time switches), such as *#ifdef*, feature toggles can change the behaviour of a running system. Our work contributes the feature toggle view which gives a new perspectives to the feature architecture of a system.

Although there are feature toggle libraries [FeatureFlags (2018)], toggles do not require specialized frameworks or new programming languages. For example, below we see the conditional statements used to guard a block of C++ code that implements "*text input focus*" feature associated with the toggle *kEnableTextInputFocusManager* in Google Chrome. If the toggle is enabled, the method *IsTextInputFocusManagerEnabled* returns true and a manager is used to deal with text focus, otherwise, a simple text client is used. With feature toggles, all features are compiled and live with a toggle configuration file or database controlling which features are available to which users.

```
bool IsTextInputFocusManagerEnabled() {
 return CommandLine::ForCurrentProcess()->HasSwitch(
      switches::kEnableTextInputFocusManager
 );
}

...

if(switches::IsTextInputFocusManagerEnabled()){
    ui::TextInputFocusManager::GetInstance()
    ->FocusTextInputClient(
        &text_input_client
    );
}else{
    input_method->SetFocusedTextInputClient(
        &text_input_client
    );
}
```

To study the effect of toggles on system architecture requires the extraction of multiple architecture views across a large system. As a result, in this work, we have chosen to examine a single project in detail. We feel that Google Chrome is a representative project because Chrome uses feature toggles in the same way as toggles are used in other internal Google code bases [Laforge (2011)]. Our detailed case study of Chrome allows us to understand and visualize how feature toggles affect the system architecture. We call this the feature toggle architectural view. Our scripts and data are available to researchers and practitioners who want to extract this new architectural view [Rahman (2017)].

1.2 Extracted Architectural Representations

The major contributions in this paper are the extraction of four separate architectural representations: the conceptual architecture, the concrete architecture, the browser reference architecture, and the feature toggle architecture.

1. Conceptual Architecture: The conceptual architecture describes the major entities and relationships among those entities. Although there is extensive documentation for Chrome, we did not find a overall architectural representation. Examining this documentation, we derive a conceptual architecture for Chrome in Section 4.

2. Concrete Architecture: The concrete architecture is a modular architecture that is based on the source code. We manually mapped over 28k source files to their respective concepts to understand the modules present in Chrome. Using the variable and function call dependencies we assigned edges between these modules. We then compared the edges and modules found in the concrete architecture with the conceptual modules. We describe discrepancies in the documented conceptual modules and edges with those that actually exist in the code. See section 5.

3. Browser Reference Architecture: A reference architecture contains the components required for a particular system domain. In 2005 Grosskurth and Godfrey extracted a browser reference architecture based on the Mozilla and Konqueror web browsers. We update this 12 year old reference architecture in Section 6.

4. Feature Toggle Architecture: We extract the *feature toggle architecture* of Chrome and illustrate the new couplings and relationships that are revealed by this new perspective (Section 7). The feature toggle architecture can be viewed from a feature's or a module's perspective. From a feature's perspective the architecture shows which modules a feature spans, see Section 7.1. From a module's perspective the architecture shows which features are contained in a module, see Section 7.2. Since features necessarily span multiple modules, we discuss the new feature relationships among modules that the toggle architecture reveals. We also discuss the evolution of the feature toggle architecture across four releases in Section 8.

Our study shows that the feature toggle architecture can give new perspectives to the feature architecture of a system. We note that the feature toggle architectural does not replace other architectures or architecture extraction methods such as feature-oriented development or cross-cutting features in aspect-oriented development. In contrast, the goal of this work is to show how the widely-used practice of feature toggles allow for a new perspective on the architecture from the feature's perspective extracted from the source code. Our contribution is to make both researchers and practitioners working on projects with feature toggles aware of the impact of toggles on the system's architecture.

2 Background on Toggles and Google Chrome

Feature toggles are simple conditionals that guard blocks of code allowing developers to enable and disable features [Rahman et al (2016)]. This mechanism facilitates dark launches [Rahman et al (2015)], disabling features, and A/B testing during rapid releases [Adams and McIntosh (2016)]. From a practical perspective, features require the following [Fowler (2010)]:

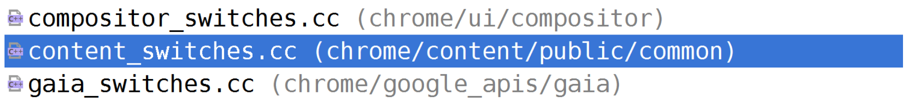
1. A configuration file with all feature toggles and their state (value).
2. A mechanism to switch the state of the toggles (for example, at program start-up or while the application is running), effectively enabling or disabling a feature.
3. Conditional blocks that guard all entry points to a feature such that changing a toggle's value can enable or disable the feature's code.

Toggles have been used in Chrome since release 5.0 in 2010. Instead of having a single configuration file for feature toggles, Chrome has multiple "switch files" containing toggles for a relevant set of features *i.e.* "feature set" (see Figure 1a). For example, there is a feature set that contains toggles to manage "Content" related features (`content_switches.cc`) which includes the feature toggle *kDisableFlashFullscreen3d*.

In Chrome, feature toggles are string constants that can be set or unset. For example, Figure 1b shows the definition of the toggle *kDisableFlashFullscreen3d*. When activated, this toggle disables the feature that renders graphics in 3D when flash content is presented full-screen. Advanced users might use this toggle to improve performance [Bravo (2012)]. These feature toggles are then used inside conditional statements, as shown in Figure 1c. If the toggle *kDisableFlashFullscreen3d* has been set a `return` statement exits the method without executing the 3D rendering feature.

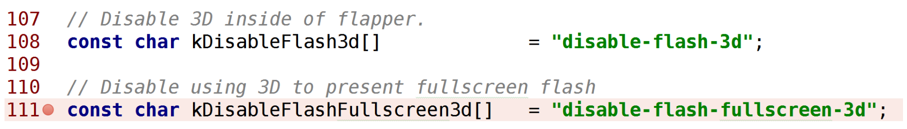
2.1 Chrome Data

To study the feature toggle architectural view, we collect the feature toggles for 30 consecutive release versions that cover more than five years of the development history of Google Chrome to understand the use of toggles on the system overtime. We study four separate release version: 5.0, 13.0, 22.0, and 34.0. We selected these releases because they cover the entire observing time period keeping an almost equal gap between each of the release versions. The versions contain 7K, 11K, 17K, and 28K C/C++ files and 1.0M, 1.4M, 2.2M and 3.7M lines of code respectively (Table 1). Before conducting our analysis, we discarded any third-party code and external plugins that were stored in the Chrome repository.



```
compositor_switches.cc (chrome/ui/compositor)
content_switches.cc (chrome/content/public/common)
gaia_switches.cc (chrome/google_apis/gaia)
```

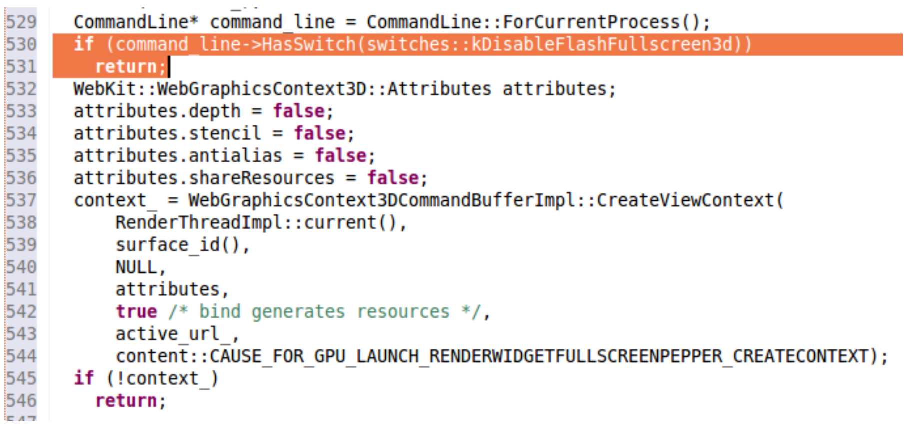
(a)



```
// Disable 3D inside of flapper.
const char kDisableFlash3d[]             = "disable-flash-3d";

// Disable using 3D to present fullscreen flash
const char kDisableFlashFullscreen3d[]    = "disable-flash-fullscreen-3d";
```

(b)



```
CommandLine* command_line = CommandLine::ForCurrentProcess();
if (command_line->HasSwitch(switches::kDisableFlashFullscreen3d))
  return;
WebKit::WebGraphicsContext3D::Attributes attributes;
attributes.depth = false;
attributes.stencil = false;
attributes.antialias = false;
attributes.shareResources = false;
context_ = WebGraphicsContext3DCommandBufferImpl::CreateViewContext(
    RenderThreadImpl::current(),
    surface_id(),
    NULL,
    attributes,
    true /* bind generates resources */,
    active_url_,
    content::CAUSE_FOR_GPU_LAUNCH_RENDERWIDGETFULLSCREENPEPPER_CREATECONTEXT);
if (!context_)
  return;
```

(c)

Fig. 1: Examples of Chrome toggles: (a) the sets of toggles composing a related feature set contained in "switch" files, (b) setting the toggle value inside `content_switches.cc`, and (c) code that is covered by the toggle *kDisableFlashFullscreen3d*.

3 Architectural Extraction Process

In Figure 2, we show the architectural extraction stages for the four architectures: the conceptual, concrete, reference, and feature toggle architecture.

3.1 Conceptual Architecture

Google Chrome does not have an existing overall conceptual diagram describing the relationships between entities in the system. We derive the conceptual architecture from the existing documentation for Google Chrome including the web resources and official wiki [Chen and Rajlich (2000),ChromeWiki (2015),Chrome (2015a),Chrome (2015b), Chrome (2016a),Chrome (2016b),Chrome (2016c)]. During a course on software architecture, a team of four graduate students, lead by the lead author and supervised by the third author, manually read

Table 1: Google Chrome Source Code Versions

Rel. Date	**#Dirs**	**#Files c, cc**	**#Feature Sets**	**#Toggles**	**Size MLOC**	**#Files Using Toggles**	**%Files Using Toggles**
5.0 May 21, 2010	1,455	7,213	6	272	1.0	4,550	63.08
13.0 Aug 2, 2011	2,206	10,886	11	466	1.4	4,992	45.85
22.0 Jul 31, 2012	3,400	17,144	21	726	2.2	10,927	63.73
34.0 Apr 8, 2014	5,411	28,585	33	1,031	3.7	12,545	43.88

and discussed the role of the each modular entity and the relationships among them. Understanding the documentation and creating an initial diagram took the team 2 person-weeks. The resulting architectural diagram is Figure 3 in Section 4.

3.2 Concrete Architecture

Table 2: Sample of File to Directory to Conceptual Module Mappings

File	Directory	Conceptual Module
`shelf_widget.cc`	`ash\shelf`	`Browser View`
`tabs_api.cc`	`chrome\...\extensions\tabs`	`Extensions`
`avatar_menu.cc`	`chrome\...\profiles`	`Data Persistence`

The concrete architecture is based on the source code files. We map the directories to the conceptual modules and then extract the call graph to the relationships between the modules. The steps are as follows:

1. **Map concrete directories to conceptual modules:**
   Following Bowman et al (1999) technique, we extract the source code and map the files and directories to modules based on the modular entities extracted from the documentation in the conceptual architectural extraction stage. Each source file is examined for its purpose based on its location in the system, name and naming conventions, source code comments, and any documented knowledge.
   This manual mapping was performed by a group of four graduate students during a course on architectural extraction. The the first author lead the

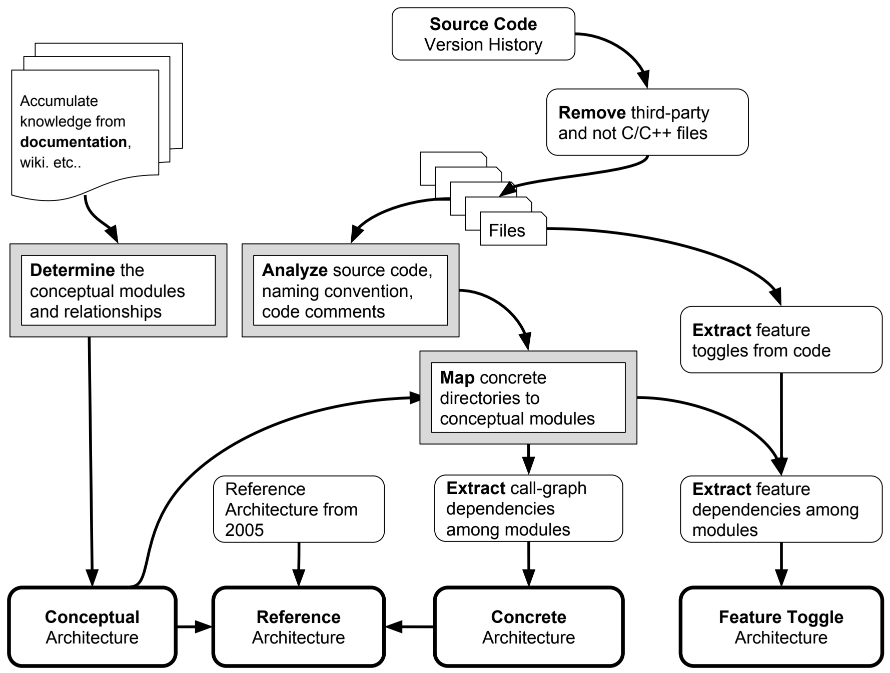

Source Code
Version History
Remove third-party and not C/C++ files
Accumulate knowledge from documentation, wiki. etc..
Files
Determine the conceptual modules and relationships
Analyze source code, naming convention, code comments
Extract feature toggles from code
Map concrete directories to conceptual modules
Reference Architecture from 2005
Extract call-graph dependencies among modules
Extract feature dependencies among modules
Conceptual Architecture
Reference Architecture
Concrete Architecture
Feature Toggle Architecture

Fig. 2: Extraction steps for conceptual, concrete, reference, and feature toggle architectural extraction. Square double boxes represent steps that required substain human input, while curved boxes represent semi-automated steps.

team and the third author supervised the project. For release 5.0, 13.0, 22.0, and 34.0 the students examined 7.2k, 10.8k, 17k, 28.6k, respectively. In total the mapping took 4 person-weeks.
The output of this stage is a mapping between conceptual modules and directories. Table 2 provides an example of these mappings with the the full set of mapping available in the replication package [Rahman (2017)].

2. **Extract the function call and variable dependencies among modules:**
For each of the source files, we extract the function call and the variable dependencies using the "Understand" Scitools (2016) tool. Combining the dependency graph and the module mapping list, we obtain the relationships between modules.
3. **Construct the concrete architecture:**
Instead of using *lsedit* like Bowman et al (1999), we use an online graph editing tool [CSAcademy (2017)] that generates directed graphs from the module-module relations. We then, normalize the inter-module relations to construct a readable concrete architecture. We removed the modules Widget, Skia, Installer, Tools, Downloads, Services, Test, Debug and Remote Host, as they are not the core to the functioning of the browser. We do not show

the edges for the `Base` and `Utility` modules as more than 46K dependencies to `Base` made by 48 modules and 2.8K dependencies to `Utility` made by 20 modules. We label the edges with the number of dependencies. Figure 4 in Section 5 shows the concrete architecture for Chrome version 34. The full list of dependencies among modules is available in our replication package [Rahman (2017)].

3.3 Browser Reference Architecture

A reference architecture contains the fundamental entities and relationships among them for a particular domain. It serves as a template to understand the system architecture in a domain. We update the 12 year old browser reference architecture that was proposed by Grosskurth and Godfrey (2005) with the extracted modern conceptual and concrete architectures of Google Chrome. New modules are added based on technology and design changes. We also verify that conceptually these modules have been adopted by other browsers, including Firefox and Safari. Figure 5 in Section 6 shows the browser reference architecture.

3.4 Feature Toggle Architecture

Table 3: Sample of Module to Feature Set Mappings

Module	Feature Set
`Browser View`	`chrome_switches`
`Browser View`	`ui_base_switches`
`Browser View`	`ash_switches`
`Content`	`base_switches`
`Content`	`content_switches`
`Content`	`ipc_switches`
`Content`	`ui_base_switches`
`Content`	`shell_switches`
`Render Engine`	`chrome_switches`
`Render Engine`	`gpu_switches`
`Render Engine`	`gl_switches`

The feature toggle architecture represents the feature toggle dependencies among the modules. Feature toggles are embedded in the source code files. Once we identify the feature toggles, we can assign feature dependencies among concrete and conceptual modules. The steps are shown on the right side of Figure 2 and discussed below.

1. **Extract feature toggles from code:**
   We extract all the feature sets ("**_switches.cc*" files) and the toggles within those files (for example, *kDisableTranslate* in translate_switches.cc) from

Table 4: Sample of Feature Set to Module Mappings

Feature Set	Module
`chrome_switches`	`Render Engine`
`chrome_switches`	`Browser Engine`
`chrome_switches`	`Browser View`
`chrome_switches`	`Apps`
`chrome_switches`	`Data Persistence`
`chrome_switches`	`UI`
`chrome_switches`	`Network`
`chrome_switches`	`Chrome OS`
`chrome_switches`	`Print`
`ui_base_switches`	`Browser View`
`ui_base_switches`	`Content`
`ui_base_switches`	`Browser Engine`
`ui_base_switches`	`Utility`
`ash_switches`	`Browser View`
`ash_switches`	`ASH`
`ash_switches`	`UI`

the source code. We drop all the files that do not depend on any feature toggle and keep only the files that use at least one feature toggle.

2. **Extract feature dependencies among modules:**
   Using the directory-to-module mapping developed for the concrete architecture we can we know which file belongs to which module. Modules that contain feature toggles from the same feature set, have a feature dependency among them. Similarly, from the features perspective we can see which modules a feature spans. We generate two mapping lists for module-feature and feature-module dependencies as the examples shown in Table 4 and 3. The full list of feature and module mappings can be found in our replication package [Rahman (2017)].
3. **Construct feature toggle architecture:**
   Using the graph editing tool [CSAcademy (2017)] we generate the graphical representation of the dependencies between modules and features, see Figure 6 in section 7.

4 Conceptual Architecture

The conceptual architecture is derived from the Google Chrome development documentation and describes the conceptual entities and relationships in the browser. The architectural extraction process we followed is described in Section 3.1. Figure 3 shows the conceptual architecture of Google Chrome. In this figure, the solid square boxes represent the modules and the edges indicate the dependencies between them. The "`Utility`", "`Base`" and "`Initializer`" modules have relationships with all modules. From the Chrome developer resources we identified 19 conceptual modules and 21 edges between those modules. We discuss the major modules below:

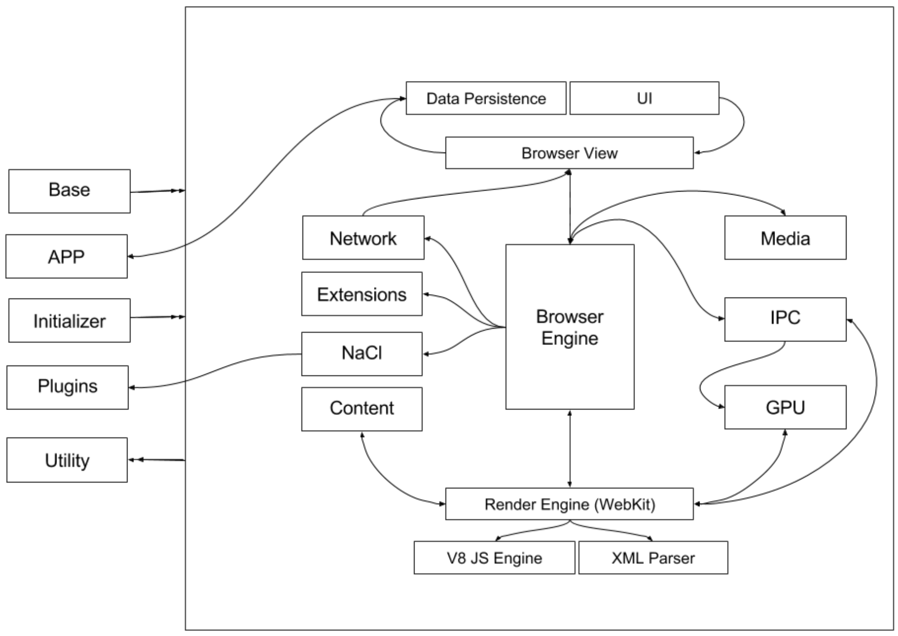


Fig. 3: Conceptual architecture of Google Chrome.

The **UI** module is the front end of the Google Chrome web browser that displays the rendered HTML with CSS, Text, Videos, and Images. The `UI` also provides developer tools, including event statistics for a page and the ability to inspect the HTML and Javascript elements.

The **Browser View** or UI Backend controls the the views that makeup the `UI` module, including the tabs, toolbar, and bookmarks bar. It represents these views in the DOM tree. The Browser View collects the rendered contents from the `Render Engine` to create the DOM tree.

The **Data Persistence** module stores data on the user's local machine to create a more efficient browsing experience. For example, browsing history, auto-fill, passwords, user preferences are stored locally so that they can be quickly accessed by the views in `Browser View`.

The three modules discussed above handle the display part of the browser. While the main processing of the web page information is done by the `Browser Engine` and `Render Engine` with the help of other modules such as `Content`, `GPU`, `NaCl`.

The **Browser Engine** is a communicator between `Browser View` and the `Render Engine`. It is the core of the browser which bridges the user interface and rendering engine. It initiates all web page events, including loading, form submission, reloading, and navigation.

The **Render Engine** renders the web page from a URL. It interprets marked up content (such as HTML, XML, and image files) and formatting information (such as CSS, XSL) with the help of the `Layout Engine`. It shares the graphic data with `GPU` to be processed separately. The `Render Engine` uses the **V8 JavaScript Engine** and **XML Parser** to process Javascript and XML contents.

The **Content** module implements the core Chrome rendering features. It also contains factory classes to create the necessary WebKit objects for Chrome. The goal of this module is to clearly separate the rendering code from other browser features.

The **GPU** provides secured access to the system's 3D Graphic APIs [Chrome (2015a)] and accelerates the processing of the graphical contents. The `Render Engine` module uses `GPU` to process and render the graphic elements in a web page.

The **IPC** module allows each tab to run as its own process facilitating interprocess communication with the core browser.

The **Extensions** module serves as a point to plug-in third-party features to enhance browser functionality. `Extensions` are managed by the `Browser Engine` [Chrome (2016c)].

NaCl is a sandbox that allows `Plugins` to run across platforms without being recompiled. It also improves the security and portability of the native Chrome code [Chrome (2016a)].

The **Network** module interfaces with the hardware to allow network connections.

5 Concrete Modular Architecture

The concrete architecture is derived from manually mapping the source code directories to the conceptual modules. Since there are over 5.4K directories in Chrome release 34.0, a team of 4 graduate students took 4 person-weeks to manually map each directory to a corresponding conceptual module. After this mapping is complete, the function call and variable dependencies are extracted using "Understand" to create the relationships between modules. Finally, the concrete diagram is drawn using a graph editing tool to improve readability. The details of the architectural extraction process is described in Section 3.2.

The concrete architecture is shown in Figure 4. We identified five additional modules, which were not present in the conceptual architecture: `Printing`, `Cloud Printing`, `Component`, `PPAPI` and `Base View`. None of these modules were explicitly discussed as separate entities in the documentation; however, when we examined the source code, the files and related directories were separate from the directories that mapped to the conceptual modules. We briefly describe the rationale for adding these additional modules. The `Printing` and `Cloud Printing` modules clearly have separate directories, but are not discussed in the documentation. The Component module implements the components that are then used in the Browser View, but resides outside of the concrete `Browser View` directory. Although conceptually the flash plugin belongs in the `Plugins` component, the implementation of this plugin, `PPAPI`, is in its own

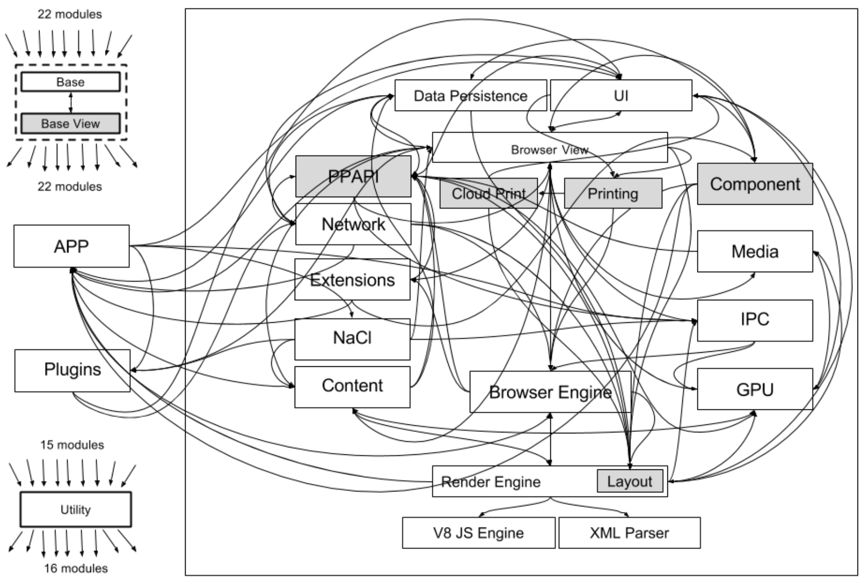


Fig. 4: Concrete architecture for Chrome Release 34.0. The gray edges represent relationships between modules that did not exist in the conceptual architecture.

directory and is tightly coupled to other parts of the system. The `Base View` was re-factored out of the `Base` module in Chrome version 13.0 to delimit the view components in the base.

The concrete architecture has 71 edges compared to the 21 edges found in the conceptual architecture. The full list of edges and the number of total variable and function calls can be found in our replication package [Rahman (2017)]. Since there are 53 new edges, we cannot discuss them all. Instead, we discuss some interesting architectural violations that we discovered in Chrome.

For example, the `Network` module has five new dependencies with `Layout`, `PPAPI`, `UI`, `App` and `Plugins`. When we examined these unexpected dependencies, we discovered that developers avoided existing interfaces. Instead, they called methods from the `Network` module directly.

The conceptual architecture contains an edge between the `GPU` and `NaCl` modules, which is missing from the concrete architecture. On investigation we found that the `GPU` receives the serialized system calls from `NaCl` via a ring buffer in shared memory. The `GPU` then parses the serialized commands and executes them with appropriate graphics calls.

A similar shared memory strategy is described in the documentation whereby the `GPU` output is communicated through the `Render Engine` on the shared memory to the `Browser Engine` and `Browser View` to finally be displayed in

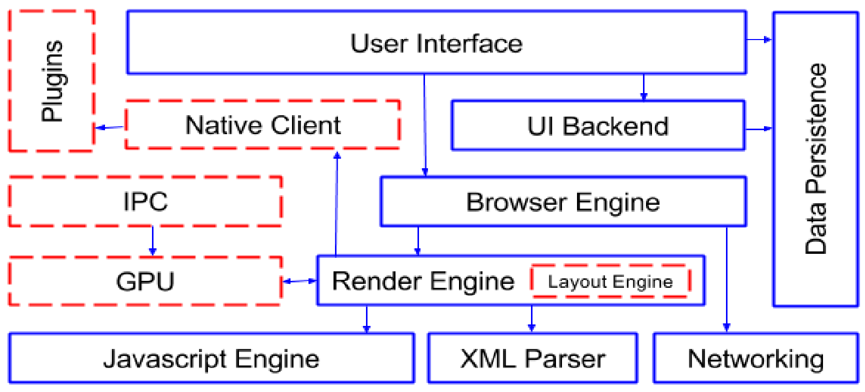


Fig. 5: Modern browser reference architecture. Boxes with red dashed borders are new components not present in the Grosskurth and Godfrey (2005) browser reference architecture.

the `UI`. However, we observe direct calls between `GPU` and the `Browser View`, and `UI` showing an unexpected direct coupling between these components.

The `Content` module exposes an API for the core rendering code. In the documentation it is unclear which modules actually use the `Content` module. The Render and Layout Engine conceptually need to call the `Content` module, however, we in the figure that there are six other modules calling the Content module.

The `Base` and `Utility` modules that contain library and other generally reusable functionality which have call relationships with 22 and 16 modules, respectively.

Bowman et al (1999) examined some of the unexpected calls found in the Linux kernel's concrete architecture. Like us they found that developers sometimes avoided interfaces to make the system more efficient. We can see this in the case of `Layout` module that bypasses interfaces in the `Browser View` to directly access `UI`.

Unlike Bowman et al (1999) who found no additional modules in the Linux concrete architecture, we found that as Chrome evolved some modules and relationships that conceptually should not exist still remain as awkward legacies. For example, Chrome is replacing the Netscape flash plugins, NPAPI, with their own Plugin (PPAPI). As the migration continues, the NPAPI continues to be in the source code but is flagged as an obsolete module [Chrome (2013)].

6 Browser Reference Architecture

From our extracted extracted conceptual and concrete architectures of Chrome, we update the 12 year old browser reference architecture that was proposed by Grosskurth and Godfrey (2005). Figure 5 shows the reference architecture. The solid boxes are the modules that are present in the 2005 reference architecture.

The red dashed boxes are modules that are new to the modern browser, including the `GPU`, `IPC`, `Plugins`, `NaCl` and `Layout Engine` modules. We discuss why each new module was included referring to other modern browsers. The extraction process is detailed in Section 3.3.

Graphical Processing Units (GPUs) have become increasingly more common. Modern web browsers, including Chrome, Firefox, Internet Explorer, Safari have the hardware acceleration module (GPU) on by default. The GPU reduces CPU consumption significantly which increases the overall performance of the of the web browser.

Multi-processor architectures have lead to a greater need for communication among asynchronous processes. Furthermore, by isolating each browser tab in a separate process problems encountered in a single tab are less likely to affect other tabs. Chrome has an Inter-Process Communication (IPC) module responsible for coordinating messages passed between process. Firefox (2015) and other popular modern web browsers adopted a similar multi-process architecture for efficient browsing.

Extending the functionality of applications with plugins has become an essential requirement of adaptable software ecosystems. Chrome has a `Plugins` module to create a clean interface for other developers to add functionality. For example, the "Microsoft Office" plugin developed by the developers at Microsoft allows Chrome users to access documents created by Microsoft Office through the Chrome browser. Chrome is not alone, all the modern web browsers can plug-in different plugins provided by other software systems, for example, Firefox has many plugins ranging from an *iTunes adapter* to *fire-bug*, *paint* and *screen capture*.

Web applications that require high performance across platforms can take advantage of the NAtive CLient `NaCl` sandbox in Chrome. This module allows the compiled C/C++ code to run with more low-level control while maintaining security and efficiently across platforms [TheRegister.CO.UK (2011)]. `NaCl` is especially useful for 3D games, CAD modelling, client-side data analytics [TheRegister.CO.UK (2011)]. Firefox has the similar native functionality provided by the module "*OdinMonkey*" [MozillaWiki (2015)].

Traditionally the `Render Engine` took the response from the `Browser Engine` and generated the tree of the HTML elements. In Chrome the task of calculating the HTML element position has been moved into a separate submodule called the `Layout Engine`. Not only Google Chrome but also Firefox introduced the Layout Engine to parse the HTML and create the frame tree for the DOM [Ethertank et al (2013)].

The intervening 12 years since 2005 browser reference architecture has seen significant technological changes, such as the introduction of GPUs. While many of the conceptual modules remain the same, both external and internal conceptual boundaries have increasingly become more defined. Our updated reference architecture provides a new guide for developers seeking a conceptual understanding of the modern web browser.

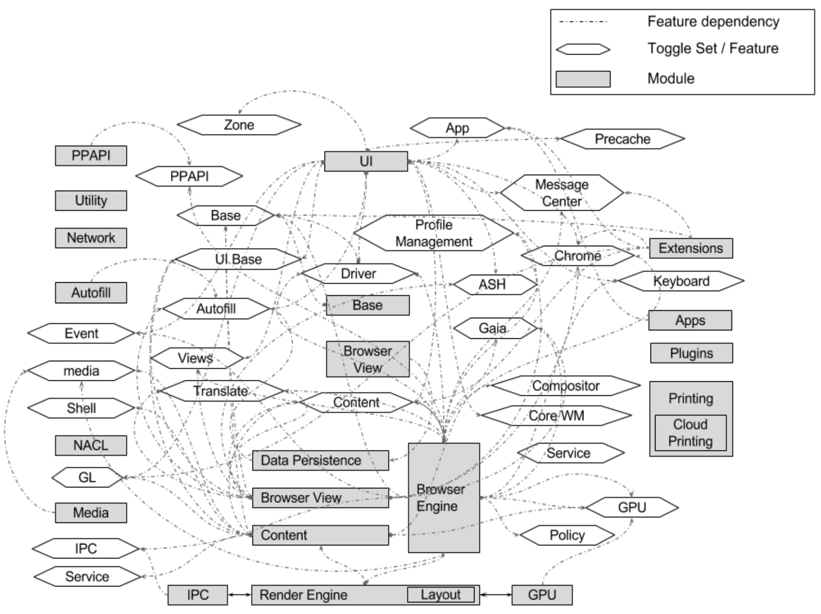


Fig. 6: Feature Toggle Achitecture

7 Feature Toggle Architecture

The use of feature toggles in the source code allows us to extract an architectural view of the features in Chrome. This helps us to identify new dependencies among modules and to better understand which modules a feature spans and which features are contained in a module. The feature architectural view helps a developer working on a feature to determine which modules will be affected by implementing a new feature or a change to an existing feature. Similarly, a developer maintaining a module will know which features he or she needs to be familiar with.

Since feature toggles are used in source code files, we can use the directory-to-module mapping derived from the concrete architecture to extract the feature toggles that affect each module. The details of the extraction process can be found in Section 3.4

Figure 6 shows the feature toggle view of the Chrome version 34. The square boxes are modules and the hexagonal boxes are named sets of features. For example, the hexagonal box "Translate" contains the feature toggles for the features related to translation, such as *kTranslateScriptURL*. Relationship edges between modules are drawn when two modules use the same feature toggle. There are 491 edges but in the interests of readability, we have chosen

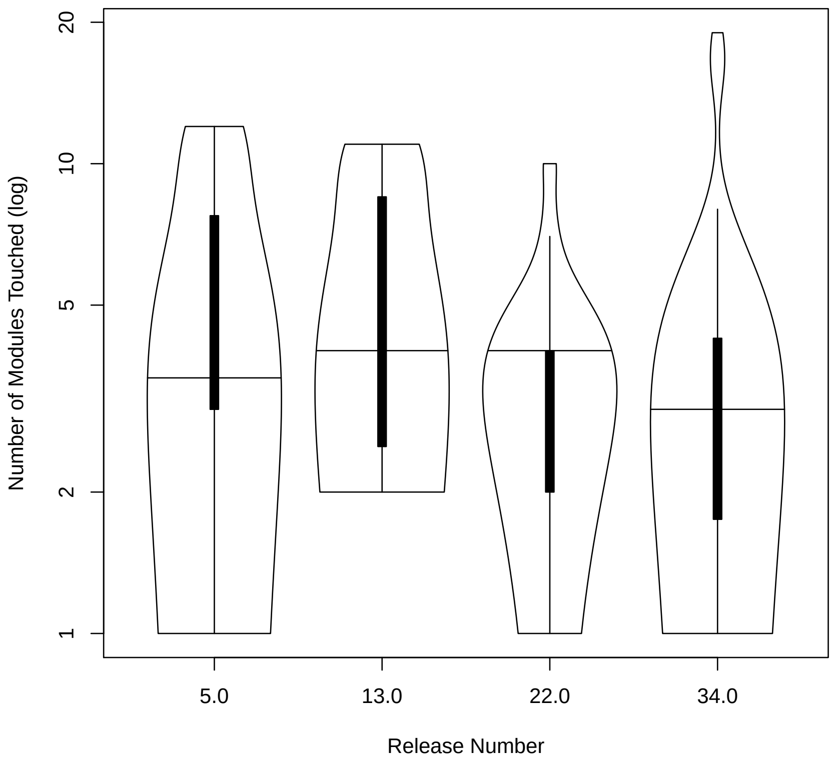


Fig. 7: The distribution of the number of modules spanned by each feature

to show only the edges that have less than 7 common feature toggles. The full list can be found in our replication package [Rahman (2017)].

The feature toggle architecture contains 1,173 edges. Compared to the concrete architecture that has 71 edges, 1,102 are new. Compared to the conceptual architecture that has 18 edges, 1,155 are new. These new edges represent natural feature dependencies which are not obvious in the traditional conceptual and concrete architectures. We discuss the feature architecture from the feature's perspective in the next and from the module's perspective in Section 7.2.

7.1 Feature's Perspective of the Feature Toggle Architecture

Features necessarily span multiple modules. For example, a “Password” feature will span at least two modules as a password must be entered in the `UI` and persisted in a data store. The feature perspective allows a developer to be aware of the modules that a feature spans. To quantify feature span, we plot the distribution of the number of modules features span in Figure 7. A box-plot also shows the quantiles of the distribution with the line representing the median. In the median case, a feature will span between three to four modules. This span implies that a developer will have to have an understanding of around four modules to fully understand a feature. Furthermore, certain features are highly complex spanning up to a maximum of 19 modules (*e.g.,* base features). Since there are more than a thousand feature toggles, we provide some illustrative examples.

The first example is the *kEnableAutoFill* which allows user information to be cached. This feature toggle spans multiple modules including the `UI`, `Browser View`, and `Data Persistence` modules to auto-populate the password, user name, and other frequent user data. This does not violate or contradict with conceptual or concrete architecture because conceptually `Browser View` has a relation with `Data Persistence` to persist user data from UI. However, from the feature view, we understand if a developer has to work on a feature such as storing password, he or she will need to touch these three modules.

The second example is the VSync feature that controls video acceleration and prevents frame-rate "stuttering" and "screen tearing" in videos. This feature is provided by the graphics library (GL) and developed under the toggle "*kEnableGpuVsync* where GPU graphic acceleration is needed. This toggle is used in `Render Engine`, `Browser Engine`, and `UI` modules to toggle code that conditionally allows the VSync feature to be enabled. Although this feature spans multiple modules, the feature dependencies are not surprising because the concrete architecture has both variable and function call relationships between the `GPU` module and the `Render Engine`, `Browser Engine`, and `UI` modules. However, the relationships between `UI` and `GPU` as well as `Browser Engine` and `GPU` are unexpected with respect to the conceptual architecture.

The third example is the Process Channel feature which is developed and controlled by the toggle "*kProcessChannelID*" that informs the child processes of the channel to listen on. As soon as a child process is spawned asynchronously by the `Browser Engine`, `Render Engine` or `GPU` modules the process channel is communicated to the `NaCl`, `App`, `Utility` and `Cloud Printing` modules. The implementation of this feature in multiple modules seems consistent with the concrete architecture as we see there are relationships between `IPC` and the modules that this feature spans. However, there are some new relationships when we consider the conceptual architecture, including those between the `IPC` and `NaCl` modules and the `Render Engine` and `Browser View` modules.

We have shown through examples, that the feature toggle architecture can allow a developer to identify the code base related to a feature. Although impact analysis may be used, our view on toggles provides the concise locations that are affected when making feature modifications.

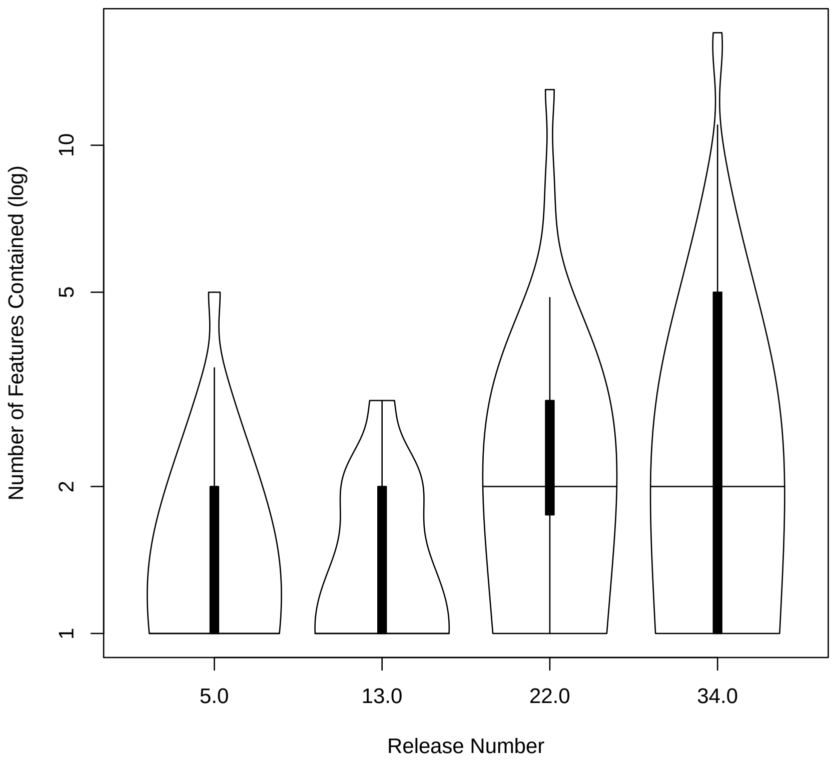


Fig. 8: The distribution of the number of features contained per module

7.2 Module's Perspective of the Feature Toggle Architecture

A developer should be aware of the features that are contained in the module in which he or she is working. We first quantify the number of features contained in each module. The distribution of features per module is shown in Figure 8. The figure also contains a boxplot imposed inside the distribution with the quantiles and a line at the median. In the median case a developer will have to understand between one to two feature sets per module. Compared with the overwhelming number of edges in the entire feature toggle architecture, we observe that a typical module contains a manageable number of features. However, we can see that over time features are being added to modules. For example, in release 34.0 the 75th percentile is at five features per module and there is one module that contains 59 features from 17 different feature sets. These modules are likely more difficult to understand and maintain. Chrome's

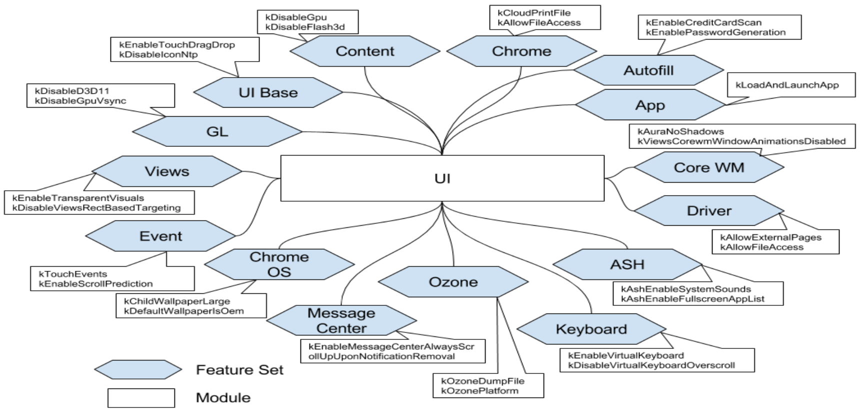


Fig. 9: UI Module containing features - Chrome Release 34.0

size does not allow us to discuss each module in detail as a result we take three examples of important modules and associated features.

The first example is the `UI` module which is responsible for interacting with the end user by providing features such as keyboard interactions and automatic form filling with user information. Figure 9 shows that the `UI` module has 17 feature sets and contains 59 feature toggles in total. The hexagonal boxes represent the feature sets while the caption boxes provide two illustrations of the feature toggles. The full list of feature toggles is available in our replication package [Rahman (2017)]. As an example, the “Event” feature set contains *kTouchEvents* which enables the touch based events. Another example, is the toggle *kCloudPrintFile* of the Chrome feature set that displays the cloud print dialogue and allows upload of files for printing.

Furthermore, the `UI` module touches many features because it is simple to disable access to the defective or incompatible features by removing the interface to the feature. For example, when the transparent “Layered Window” (also known as glass window) feature was released for Chrome browser, developers found it incompatible with the Direct3D surfaces. As a result, developers disabled this UI feature by using a release toggle *kGlassFrameOverlayAlpha*. Although the toggle architecture shows that `UI` module contains a large number of features, the conceptual architecture does not reflect this high coupling.

A second example is the `Browser Engine` which is responsible for rendering the HTML and CSS into the properly formatted visual objects. The `Browser Engine` contains the largest number of features of any module. One of the feature toggles is the *kEnableGpuShaderDiskCache* which allows the `Browser Engine` to receive rendered output from `GPU` and `Render Engine` via the shared cache memory. `Browser Engine` also contains features such as “Virtual Keyboard”, “Translate” that are controlled by *kEnableVirtualKeyboard* and *kDisableTranslate* toggles respectively. Since the `Browser Engine` is one of the core modules, it is not surprising that this module contains a multitude of features that span from other modules. We do note that some features cause violations of the conceptual architecture. For example, the feature toggles *kEnableGpuAcceleration* has been used in `Browser Engine` to control code that depends on the GPU acceleration feature. However, we do not see any direct relationship between `GPU` and `Browser Engine` in the conceptual architecture.

The third example is the `Plugins` module which provides a consistent interface for external plugin code that does not depend on the native Chrome functionality. As a simple plugin point, there are few features toggles. An example toggle is *kEnablePluginPowerSaver* which globally informs plugins to use a power saving strategy.

The examples illustrate the complex features that make up Chrome's modules and feature toggle architectural makes these feature dependencies relationships clear to developers and researchers.

8 The Effect of Feature Toggles on Architectural Evolution

As a system evolves, features and modules can become coupled leading to violations of the modular architecture. From a raw size perspective, Chrome has grown substantially. In release version 5.0 Chrome had a total of 1.4K directories and 7.2K C/C++ files. Relative to Release 5.0, releases 13.0, 22.0, and 34.0, had an increase of 3K, 7K and 11K files, respectively. At release 34.0 there are 5.4K directories and 28.5K files.

Relative to the file size increases, the number of toggles was initially growing at a similar rate. However, over the releases the rate of increase in number of toggles is slower than the rate of increase in number of files. In early releases of Google Chrome, there were very few feature toggles in use. As the project grew, the number of feature toggles grew and they spanned more modules. At release 34.0, Chrome developers initiated a cleanup of unused toggles and associated dead code [Rahman et al (2016)], which resulted in a drop in the number of total toggles and reduced feature span across modules (see Figure 7). Despite this cleanup of old toggles, many new toggles were added which explains the overall increasing trend with the total number of toggles doubling between release 22.0 to 34.0.

The number of modules spanned by a feature in release 5 and 13 were higher than the number of features contained in a module. Many features span more than 10 modules in these releases, see Figure 7. Release 22.0 and 34.0 saw an increase in both the number of modules and the number of features per module. However, features tended to span fewer modules with the a median of two to five modules per feature. Despite the growth, features tend to be more concentrated in modules.

Below we discuss examples of feature evolution from the module and feature toggles perspectives. For example, in release version 5.0, the base features used to span 6 modules such as `Browser Engine`, `Apps`, `IPC`, `Network`, `SSL` and

`NaCl`. However, in release version 22.0 the base feature set was re-factored with toggles moving to the new feature set related to IPC. This change reduced the couple of the base feature set.

A second example, is the touch screen feature that was developed under the toggle *kEnableTouch* and was introduced at Release 6.0. At Release 6.0, the toggle spanned the `UI` and `Browser View` modules. At release 17.0 Chrome developers renamed this toggle to *kEnableTouchEvents* and moved it inside the `Content` module to implement additional touch based features. Over the next few releases Chrome kept improving touch-screen compatibility by introducing more features with toggles, such as *kEnableTouchpadThreeFingerClick* at release 18.0, *kEnableTouchCallibration* at release 21.0, and *kEnableTouchDragDrop* at Release 25.0. As improvements were made the “Touch Screen” feature now spans many modules, including `Content` and `UI Base`.

A third example is the “Printing” feature, which was developed under the *kPrint* toggle. The feature spanned two modules `UI` and `Printing` before the introduction of `Cloud Printing` in Release 22.0. At release 22.0 *kPrint* toggle was renamed to *kPrintRaster* and it now also spans the `Browser View`.

Examining feature evolution from a module’s perspective, the `Content` module illustrates the large increase in feature toggles over time. At release version 13.0, the `Content` module had 113 toggles demonstrating a wide range of content related feature. The features include *kDisableAudio*, *kRendererProcess*, and *kGpuProcess* that are responsible for browser audio disabling, initiating rendering and initiating GPU process, respectively. Over time new features have been introduced to increase the responsibilities of the `Content` module to serve more functionality to the `Render Engine`. For example the *kEnableVideoTrack* toggle was added in release 22.0 to enable video tracking capability. At release 34.0 the number of toggles in `Content` module increased to 285.

Software architecture evolves over time adding new features and functionality. The traditional modular architecture can tell us about this evolution of module dependencies. However, the feature toggle architecture also allows us to understand and explain the evolution of feature dependencies.

9 Related Work

9.1 Automated Architectural Extraction

Many works have clustered the relationships among files and modules to automatically extract an architecture. Below we discuss some approaches and their limitations.

A comparative study of clustering algorithms for re-modularization of software was conducted by Anquetil and Lethbridge (2003). Clustering architectural components depends upon i) abstract dependencies of entities that will be clustered, ii) metrics for determining couples between entities and iii) the algorithm for clustering. They found that a proper description scheme is highly important for the entities or modules being clustered. They also propose a novel description scheme as well as better formal evaluation methods for clustering results. Other researchers have also investigated cluster algorithms for architectural extraction, including Maqbool and Babri (2004), Andritsos and Tzerpos (2005), Wu et al (2005), Kobayashi et al (2012), and Naseem et al (2011). Instead of automatic clustering, we use the knowledge and man-power of multiple students to manually extract related entities.

Garcia et al (2013) surveyed six automated techniques to extract and compare architectures.

1. Algorithm for Comprehension-Driven Clustering (ACDC) [Tzerpos and Holt (2000)]
2. Weighted Combined Algorithm (WCA) [Maqbool and Babri (2004)], and its two measurements WCA-UE and WCA-UENM.
3. scaLable InforMation Bottleneck [Andritsos and Tzerpos (2005)]
4. Bunch [Mancoridis et al (1999)]
5. Zone-Based Recovery [Corazza et al (2010, 2011)]
6. Architecture Recovery using Concerns (ARC) [Garcia et al (2011)]

They found that each technique produced drastically different architectural representations of the same software. Compared to a manually extracted architectural benchmark, even the best two automated approaches (ARC and ACDC) had low accuracy suggesting that manual recovery may be the best strategy. As a result, we use a semi-automated approach to extract the conceptual, concrete, reference, and feature toggle architectures.

Although automated clustering approaches exist, many works use a semi-automated strategy to extract the architecture of a system. For example, Bowman et al (1999) were the first to extract the architecture of Linux Kernel using a semi-automated technique as a combination of manual intervention and tool support. They obtained a concrete architecture by manually extracting the dependencies among the call graph, variable references, and the naming conventions of files and directories followed by the use of tools to construct the final architectural representation. We follow a similar strategy to extract Chrome's modular architecture technique. We then apply feature toggles to the architecture to understand feature relationships among modules.

9.2 Architectural Views

Kruchten et al.described a 4+1 view of the architecture [Kruchten (1995)]. The 4+1 view provides a model describing software architecture based on multiple concurrent views such as end-users view, developers view, system-engineer's view and project manager's view which resembles the logical, development, physical and process view. In Krikhaar's doctoral research [Krikhaar (1999)], he examined software architectures from additional viewpoints including Logical View, Module View, Code View, Execution View and Physical View. He considered a number of architectural views such as Logical View, Module View, Code View, Execution View and Physical View. Our study contributes a two new views of the software's feature toggle architecture: the features a module spans and the features that are contained in a module.

9.3 Feature Architectures, Product Lines, and AOP

Many works have studied the software feature architectures. However, researchers studied usually study software systems that are developed in feature oriented way or following the product-lines methodologies. In contrast, feature toggles do not require an explicit product line methodology and simply introduce live flags into the code.

Software product lines allow for the release of variations in a software system based on use cases. A product line method FeatureRSEB was developed by Griss Griss et al (1998) combining the two ideas of domain analysis (FODA) [Kang et al (1990)] and reuse based development. FeatureRSEB performs an exclusive mapping between feature, modules, and architectural elements through traceability links that are related to the use-cases of the software [Sochos et al (2004)]. Use cases ultimately point to the classes inside the system within the architectural elements which allows developers understand the mapping between the system and the various product lines. In contrast, Chrome uses a simple feature toggles approach to allow features to be mapped across multiple modules. Toggles can encapsulate any code feature and are not dependent on specific use cases allowing for greater flexibility.

Dintzner et al (2016) studied variability in the Linux Kernel. The study focused on information changes in variability models, assets, and mappings between variable features and actual assets. To track down variability they consider the *"KConfig"* files and the pre-processor based `C` code files as the assets and the "Make" files as the mapping. Compared to their work, we extract feature view of the Chrome architecture considering the toggle configuration files as the feature sets and the toggles inside those files as the representative of features.

Aspect Oriented Programming (AOP) approaches use different techniques to achieve multiple separation of concerns [Kiczales et al (2001),Batory et al (2003)]. Traditionally AOP focuses on code level (class files, UML etc.). Jansen and Bosch (2005) focus on the cross-cutting concerns of design decisions at the architectural level where features impose design decisions and the cross-cutting concerns. In contrast to Aspects, feature toggles do not require any modifications to the programming language but still allow developers to cross-cut multiple modules and turn on/off certain cross-cutting features. The feature toggle architecture allows us to identify the features within the source code and extract the feature's view of the modular architecture extracted from the source code.

9.4 Feature Location and Traceability Recovery

When a developer is working on a newly assigned task he or she needs to understand where to start implementing the new feature. Identifying an initial location for a feature within the source code is commonly known as feature location or concept location [Dit et al (2013)]. A survey on feature location by Dit et al (2013) found that "feature location" is one of the most important and common activities performed by programmers during software maintenance and evolution. There has been much research into feature location using the dependency graph generated from the source code [Chen and Rajlich (2000)], traceability link recovery between documentation [Lindvall and Sandahl (1996); Mirakhorli and Cleland-Huang (2011)], impact analysis [Queille et al (1994)], and aspect mining [Kellens et al (2007)]. The toggle architecture combined with the modular architecture allows a developer to locate feature code across modules. Looking at the feature in a particular module will inform a developer where to being making modification, *e.g.,* the UI module for a password related feature.

Identifying traceability links between source code and documentation has received considerable attention. **Information retrieval** techniques have been widely used to resolve the links between source code elements and documentation. For example, Antoniol et al (2002) apply a probabilistic and Vector Space Model (VSM) to resolve terms, while Marcus and Maletic (2003) use Latent Semantic Indexing (LSI). Unfortunately these approaches suffer from low precision and recall. Recent techniques that depend on language specific context have achieved precision and recall above 90% but are only implemented for Java [Dagenais and Robillard (2012), Rigby and Robillard (2013)] making them inappropriate for our case study of Chrome.

Feature location and recovery techniques identify features without explicit markers. In contrast, with feature toggles, developers mark the code as part of a feature. The features are then extracted explicitly instead of using statistical and other recovery techniques. We do not contribute to the advancement of feature recover, instead we note that feature toggles require developers to design the architecture of the system around isolating features so that can be toggled off and on in a live system. We use these toggles to understand how features influence the modular architecture of Chrome.

10 Threats to Validity

We studied a single project the Google Chrome web browser, so our results many not generalize to other projects. Since the extraction of the architecture of a large system takes months, the choice of a single project seems reasonable. Our methodology is applicable to any project that uses feature toggles. Our approach would be difficult to fully automate as few projects have defined a mapping between the conceptual modules and the concrete source code directories. In contrast, while the syntax for representing feature toggles will vary among projects, this extraction stage can be easily automated by adapting our scripts. The recent development of feature toggle libraries for major programming languages should ease the extraction of toggles from many projects by providing a common injection and management infrastructure [FeatureFlags (2018)].

On Chrome, feature toggles are not always permanent in the source code. For example, when a new feature is implemented the old feature and all related toggles may be removed. To partially deal with this issue, we extracted the architecture from four different release versions spanning 30 release versions (5.0 to 34.0) in Chrome history. Since toggles are grouped into sets, it is unlikely that the entire feature set will be removed, so we should still have one dependency among the modules which is with the toggles set itself. We may lose some granularity of features because we study four release versions. The more release versions we observe the more accurate information about features and their nature of expanding into modules we explore. However, this need for exploring additional release versions must be balanced against the manual effort.

We considered only call and feature toggle dependencies. Future work could examine other types of dependency, such as compilation dependency.

The mapping of concepts in the documentation to source files and directories was performed as part of a class project by four graduate students and lead by the first author and supervised by the third author. Like any human activity, this process is prone to human error since manual effort is involved. To alleviate the threat due to human error, we had multiple people work on the mapping and resolve inconsistencies through discussion.

11 Conclusion

In conclusion, we extracted four architectures for Google Chrome: the conceptual, concrete, browser reference, and feature toggle architectures. The extraction involved examining the conceptual modular entities in the documentation, the variable reference and function call relationships, the modules that are common to multiple modern browsers, and the feature toggle relationships among modules. We also examined the evolution of the feature toggles architecture over time. As can be seen in Figure 2, the extracted architectures are derived from different but related sources and serve to triangulate each other as the concluding discussion below shows.

Google Chrome did not have an existing conceptual diagram describing the relationships between system entities, so we derived one by examining and discussing the documentation and online forums. Figure 3 shows the 19 conceptual modules and 21 edges we identified, including the Browser View, Data Persistence, Browser Engine, and Render Engine modules. This conceptual diagram shows the entire architecture of Chrome and indicates a clear cohesiveness and separation of modules and relationships.

The concrete architecture describes the call graph relationships of the source code between modules. We derive it by manually mapping each source file and directory to those found in the conceptual architecture. This manual mapping took 4 person-weeks. We then extracted the call graph from the source files to determine relationships among modules. In the resulting architectural diagram in Figure 4 we identified 5 modules not discussed in the Chrome documentation and 24 modules in total with 72 call edges. We noted some interesting source code violations of the conceptual architecture. Some violations bypassed interfaces to make the system more efficient, while others were related to legacy code, such as the development work to replace the aging Netscape flash plugin.

The browser reference architecture describes the modules required in a modern web browser. We derived this architecture from the conceptual, concrete, and 12 year old [Grosskurth and Godfrey (2005)] reference architecture. In Figure 5 we see that while many of the major modules remain the same, modern browsers have a sophisticated plugin module to allow external developers to create add-ons. Furthermore, technological changes, such as GPUs, have introduced new modules.

The feature toggle architecture is extracted from source code and shows which features affect which modules and vice versa. Feature toggles are named ‘switches’ that cover the code that implements a feature. They are used by large web companies to quickly enable and disable a feature in production [Rahman et al (2016)]. Using the directory-to-module mapping derived for the concrete architecture, we extract the toggles contained in each module. Toggles that are contained in multiple modules indicate a feature relationship between these modules. In Figure 6 we see that there are 1,173 feature relationships between modules. These feature dependencies do not necessarily indicate modular architectural violations as features must span multiple modules. For example, the “EnableAutoFill” feature toggle must be present in the UI module to allow the user to select the content to autofill and also be present in Data Persistence module to store potential autofill content. Our novel contribution of extracting the feature toggle architecture allows researchers and developer to view the features that are present in a module and the modules that are necessary for a feature.

This work has many audiences. Those interested in teaching or working on web browser technology now have a reference to the major architectural modules as well as an case study of one of the most popular web browsers. Developers and researchers who are working on or studying systems with feature toggles now have a method and technique to view the impact of toggles on the modular architecture of the system. Finally, this work required substantial manual effort to extract multiple architectures, we hope that future researchers will use our replication package [Rahman (2017)] and file to module mappings to test their automated architectural extraction tools and techniques.

Acknowledgements

We would like to thank Sumit Sarkar, Alexey Moujeer, Taiseem Ahmed and Gautam Sidharth Verma for their assistance in mapping architectural entities in the documentation to the concrete source files and directories.

References

Adams B, McIntosh S (2016) Modern release engineering in a nutshell–why researchers should care. In: Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on, IEEE, vol 5, pp 78–90

Andritsos P, Tzerpos V (2005) Information-theoretic software clustering. IEEE Trans Softw Eng 31(2):150–165, DOI 10.1109/TSE.2005.25, URL http://dx.doi.org/10.1109/TSE.2005.25

Anquetil N, Lethbridge TC (2003) Comparative study of clustering algorithms and abstract representations for software remodularisation. IEE Proceedings - Software 150(3):185–201, DOI 10.1049/ip-sen:20030581

Antoniol G, Canfora G, Casazza G, De Lucia A, Merlo E (2002) Recovering traceability links between code and documentation. IEEE Transactions on Software Engineering 28(10):970–983

Batory D, Liu J, Sarvela JN (2003) Refinements and multi-dimensional separation of concerns. In: Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, New York, NY, USA, ESEC/FSE-11, pp 48–57, DOI 10.1145/940071.940079, URL http://doi.acm.org/10.1145/940071.940079

Benavides D, Trinidad P, Ruiz-Cortés A (2005) Automated reasoning on feature models. In: Proceedings of the 17th International Conference on Advanced Information Systems Engineering, Springer-Verlag, Berlin, Heidelberg, CAiSE'05, pp 491–503, DOI 10.1007/11431855_34, URL http://dx.doi.org/10.1007/11431855_34

Bowman IT, Holt RC, Brewster NV (1999) Linux as a case study: Its extracted software architecture. In: Proceedings of the 21st International Conference on Software Engineering, ACM, New York, NY, USA, ICSE '99, pp 555–563, DOI 10.1145/302405.302691, URL http://doi.acm.org/10.1145/302405.302691

Bravo (2012) List of chromium command line switches (for 2012-09-26). http://peter.sh/experiments/chromium-command-line-switches/?date=2012-09-26

Chen K, Rajlich V (2000) Case study of feature location using dependence graph. In: Proceedings IWPC 2000. 8th International Workshop on Program Comprehension, pp 241–247, DOI 10.1109/WPC.2000.852498

Chrome (2015a) The chromium projects: Design documents. http://www.chromium.org/developers/design-documents

Chrome (2015b) Google chrome developers' documentation. https://developer.chrome.com/extensions/devguide

Chrome (2016a) The chromium projects: Nacl and pnacl. http://www.chromium.org/native-client/nacl-and-pnacl

Chrome (2016b) Content module. http://www.chromium.org/developers/content-module

Chrome (2016c) What are extensions. https://developer.chrome.com/extensions

Chrome G (2013) Saying goodbye to our old friend npapi. http://bit.ly/2fgG8UX

ChromeWiki (2015) Google chrome official wiki page. http://en.wikipedia.org/wiki/Chrome

Corazza A, Di Martino S, Scanniello G (2010) A probabilistic based approach towards software system clustering. In: Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering, IEEE Computer Society, Washington, DC, USA, CSMR '10, pp 88–96, DOI 10.1109/CSMR.2010.36, URL http://dx.doi.org/10.1109/CSMR.2010.36

Corazza A, Martino SD, Maggio V, Scanniello G (2011) Investigating the use of lexical information for software system clustering. In: Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering, IEEE Computer Society, Washington, DC, USA, CSMR '11, pp 35–44, DOI 10.1109/CSMR.2011.8, URL http://dx.doi.org/10.1109/CSMR.2011.8

CSAcademy (2017) Graph editor. https://csacademy.com/app/graph_editor

Czarnecki K, Helsen S, Eisenecker U (2004) Staged Configuration Using Feature Models, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 266–283. DOI 10.1007/978-3-540-28630-1_17, URL https://doi.org/10.1007/978-3-540-28630-1_17

Dagenais B, Robillard MP (2012) Recovering traceability links between an API and its learning resources. In: Proceedings of the 34th ACM/IEEE International Conference on Software Engineering, pp 47 –57

Dintzner N, v Deursen A, Pinzger M (2016) Fever: Extracting feature-oriented changes from commits. In: 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), pp 85–96, DOI 10.1109/MSR.2016.018

Dit B, Revelle M, Gethers M, Poshyvanyk D (2013) Feature location in source code: a taxonomy and survey. Journal of software: Evolution and Process 25(1):53–95

Dixon E, Enos E, Brodmerkle S (2011) A/b testing of a webpage. US Patent 7,975,000

Dubinsky Y, Rubin J, Berger T, Duszynski S, Becker M, Czarnecki K (2013) An exploratory study of cloning in industrial software product lines. In: Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering, IEEE Computer Society, Washington, DC, USA, CSMR '13, pp 25–34, DOI 10.1109/CSMR.2013.13, URL http://dx.doi.org/10.1109/CSMR.2013.13

Ethertank, DBaron, Kohei, Kennykaiyinyu (2013) https://mzl.la/2i7M1vp

FeatureFlags (2018) Feature flags, toggles, controls: Libraries/sdks. https://featureflags.io/feature-flags/

Firefox (2015) Multi-process firefox - technical overview. https://mzl.la/2AwsuJp

Fitzgerald B, Stol KJ (2014) Continuous software engineering and beyond: Trends and challenges. In: Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering, ACM, New York, NY, USA, RCoSE 2014, pp 1–9, DOI 10.1145/2593812.2593813, URL http://doi.acm.org/10.1145/2593812.2593813

Fowler (2010) Featuretoggle. http://martinfowler.com/bliki/FeatureToggle.html

Garcia J, Popescu D, Mattmann C, Medvidovic N, Cai Y (2011) Enhancing architectural recovery using concerns. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society, Washington, DC, USA, ASE '11, pp 552–555, DOI 10.1109/ASE.2011.6100123, URL http://dx.doi.org/10.1109/ASE.2011.6100123

Garcia J, Ivkovic I, Medvidovic N (2013) A comparative analysis of software architecture recovery techniques. In: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering, IEEE Press, Piscataway, NJ, USA, ASE'13, pp 486–496, DOI 10.1109/ASE.2013.6693106, URL https://doi.org/10.1109/ASE.2013.6693106

Griss ML, Favaro J, Alessandro Md (1998) Integrating feature modeling with the rseb. In: Proceedings of the 5th International Conference on Software Reuse, IEEE Computer Society, Washington, DC, USA, ICSR '98, pp 76–, URL http://dl.acm.org/citation.cfm?id=551789.853486

Grosskurth A, Godfrey MW (2005) A reference architecture for web browsers. In: Proceedings of the 21st IEEE International Conference on Software Maintenance, IEEE Computer Society, Washington, DC, USA, ICSM '05, pp 661–664, DOI 10.1109/ICSM.2005.13, URL http://dx.doi.org/10.1109/ICSM.2005.13

Jansen A, Bosch J (2005) Software architecture as a set of architectural design decisions. In: Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture, IEEE Computer Society, Washington, DC, USA, WICSA '05, pp 109–120, DOI 10.1109/WICSA.2005.61, URL https://doi.org/10.1109/WICSA.2005.61

Kang K, Cohen S, Hess J, Novak W, Peterson S (1990) Feature–Oriented Domain Analysis (FODA) Feasibility Study. Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University

Kellens A, Mens K, Tonella P (2007) A survey of automated code-level aspect mining techniques. Transactions on aspect-oriented software development IV pp 143–162

Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold WG (2001) An overview of aspectj. In: Proceedings of the 15th European Conference on Object-Oriented

Programming, Springer-Verlag, London, UK, UK, ECOOP '01, pp 327–353, URL http://dl.acm.org/citation.cfm?id=646158.680006

Kobayashi K, Kamimura M, Kato K, Yano K, Matsuo A (2012) Feature-gathering dependency-based software clustering using dedication and modularity. In: Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM), IEEE Computer Society, Washington, DC, USA, ICSM '12, pp 462–471, DOI 10.1109/ICSM.2012.6405308, URL http://dx.doi.org/10.1109/ICSM.2012.6405308

Krikhaar RL (1999) Software architecture reconstruction. Philips Electronics

Kruchten P (1995) Architecture Blueprints&Mdash;the &Ldquo;4+1&Rdquo; View Model of Software Architecture. TRI-Ada '95, ACM, New York, NY, USA, DOI 10.1145/216591.216611, URL http://doi.acm.org/10.1145/216591.216611

Laforge (2011) Chrome release cycle. http://www.slideshare.net/Jolicloud/chrome-release-cycle, job title: Technical Program Manager (Chrome) at Google

Lindvall M, Sandahl K (1996) Practical implications of traceability. Softw Pract Exper 26(10):1161–1180, DOI 10.1002/(SICI)1097-024X(199610)26:10⟨1161::AID-SPE58⟩3.3.CO;2-O, URL http://dx.doi.org/10.1002/(SICI)1097-024X(199610)26:10<1161::AID-SPE58>3.3.CO;2-O

Mancoridis S, Mitchell BS, Chen Y, Gansner ER (1999) Bunch: A clustering tool for the recovery and maintenance of software system structures. In: Proceedings of the IEEE International Conference on Software Maintenance, IEEE Computer Society, Washington, DC, USA, ICSM '99, pp 50–, URL http://dl.acm.org/citation.cfm?id=519621.853406

Maqbool O, Babri HA (2004) The weighted combined algorithm: A linkage algorithm for software clustering. In: Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04), IEEE Computer Society, Washington, DC, USA, CSMR '04, pp 15–, URL http://dl.acm.org/citation.cfm?id=977397.977725

Marcus A, Maletic J (2003) Recovering documentation-to-source-code traceability links using latent semantic indexing. In: Proceedings of the 25th ACM/IEEE International Conference on Software Engineering, pp 125–135

Mirakhorli M, Cleland-Huang J (2011) Transforming trace information in architectural documents into re-usable and effective traceability links. In: Proceedings of the 6th International Workshop on SHAring and Reusing Architectural Knowledge, ACM, New York, NY, USA, SHARK '11, pp 45–52, DOI 10.1145/1988676.1988685, URL http://doi.acm.org/10.1145/1988676.1988685

MozillaWiki (2015) javascript:spidermonkey:odinmonkey. https://mzl.la/2A4K5YQ

Naseem R, Maqbool O, Muhammad S (2011) Improved similarity measures for software clustering. In: Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering, IEEE Computer Society, Washington, DC, USA, CSMR '11, pp 45–54, DOI 10.1109/CSMR.2011.9, URL http://dx.doi.org/10.1109/CSMR.2011.9

Queille JP, Voidrot JF, Wilde N, Munro M (1994) The impact analysis task in software maintenance: A model and a case study. In: Proceedings of the International Conference on Software Maintenance, IEEE Computer Society, Washington, DC, USA, ICSM '94, pp 234–242, URL http://dl.acm.org/citation.cfm?id=645543.655725

Rahman AAU, Helms E, Williams L, Parnin C (2015) Synthesizing continuous deployment practices used in software development. In: 2015 Agile Conference, pp 1–10, DOI 10.1109/Agile.2015.12

Rahman MT (2017) Chrome architecture replication package. https://github.com/tajmilur-rahman/chrome-architecture

Rahman MT, Querel LP, Rigby PC, Adams B (2016) Feature toggles: Practitioner practices and a case study. In: Proceedings of the 13th International Conference on Mining Software Repositories, ACM, New York, NY, USA, MSR '16, pp 201–211, DOI 10.1145/2901739.2901745, URL http://doi.acm.org/10.1145/2901739.2901745

Rigby PC, Robillard MP (2013) Discovering essential code elements in informal documentation. In: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, pp 832–841

Rubin J, Czarnecki K, Chechik M (2015) Cloned product variants: From ad-hoc to managed software product lines. Int J Softw Tools Technol Transf 17(5):627–646, DOI 10.1007/s10009-014-0347-9, URL http://dx.doi.org/10.1007/s10009-014-0347-9

Scitools (2016) Understand source code analyzer. https://scitools.com

Sochos P, Philippow I, Riebisch M (2004) Feature-oriented development of software product lines: Mapping feature models to the architecture. In: Object-Oriented and Internet-Based Technologies: 5th Annual International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World, Net. ObjectDays 2004 Erfurt, Germany, September 27–30, 2004 Proceedings, Springer, vol 3263, p 138

TheRegisterCOUK (2011) Google's native client goes live in chrome. http://www.theregister.co.uk/2011/09/16/native_client_debuts_in_chrome/

Tzerpos V, Holt RC (2000) Acdc: An algorithm for comprehension-driven clustering. In: Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00), IEEE Computer Society, Washington, DC, USA, WCRE '00, pp 258–, URL `http://dl.acm.org/citation.cfm?id=832307.837118`

Wu J, Hassan AE, Holt RC (2005) Comparison of clustering algorithms in the context of software evolution. In: Proceedings of the 21st IEEE International Conference on Software Maintenance, IEEE Computer Society, Washington, DC, USA, ICSM '05, pp 525–535, DOI 10.1109/ICSM.2005.31, URL `http://dx.doi.org/10.1109/ICSM.2005.31`