Integrated Tooling Framework for Software Configuration Analysis

by
Nieraj Singh
BA, University of Colorado, 1997

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

© Nieraj Singh, 2011
University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopy
or other means, without the permission of the author.

Supervisory Committee

Integrated Tooling Framework for Software Configuration Analysis

by

Nieraj Singh
BA, University of Colorado, 1997

Supervisory Committee

Dr. Yvonne Coady, Department of Computer Science
Supervisor

Dr. Melanie Tory, Department of Computer Science
Departmental Member

Dr. Nigel Horspool, Department of Computer Science
Departmental Member

Abstract

Supervisory Committee
Dr. Yvonne Coady, Department of Computer Science
Supervisor
Dr. Melanie Tory, Department of Computer Science
Departmental Member
Dr. Nigel Horspool, Department of Computer Science
Departmental Member

Configurable software systems adapt to changes in hardware and execution environments, and often exhibit a variety of complex maintenance issues. Many tools exist to aid developers in analysing and maintaining large configurable software systems. Some are standalone applications, while a growing number are becoming part of Integrated Development Environments (IDE) like Eclipse. Reusable tooling frameworks can reduce development time for tools that concentrate on software configuration analysis. This thesis presents C-CLEAR, a common, reusable, and extensible tooling framework for software configuration analysis, where clear separation of concern exists between tooling functionality and definitions that characterise a software system. Special emphasis will be placed on common mechanisms for data abstraction and automatic IDE integration independent of the software system that is being analysed.

Table of Contents

Supervisory Committee ii
Abstract iii
Table of Contents iv
List of Tables vi
List of Figures vii
Chapter 1: Introduction and Related Work 1
1.1 Software Configuration Problems 5
1.2 Related Tooling 8
1.2.1 Software Build Systems 9
1.2.2 Historical Analysis 11
1.2.3 Abstraction and Visualisation 12
1.3 Need for Tools for Software Analysis 15
1.4 Common Tooling Features 17
1.5 Use Case Study 19
Chapter 2: Design 25
2.1 Design Specifics 26
2.2 Use Case Explored 33
2.3 Summary 36
Chapter 3: Implementation 38
3.1 Core 39
3.1.1 Syntax Model 40
3.1.2 Domain Language Provider 43
3.1.3 Lexical Tokeniser 47
3.1.4 Syntax Parser 49
3.1.5 Tokenising and Parsing Summary 54
3.1.6 Construct Marking and Indexing 55
3.1.7 Parser Controller 57
3.2 User Interface 60
3.2.1 Project Selection 60
3.2.2 Constructs View 61
3.2.3 Query Results View 64
3.2.4 Query Preferences Page 66
3.3 Query 67
3.3.1 Domain Query Provider 68
3.3.2 Domain and Tooling Functionality Separation 73
3.4 Summary 75
Chapter 4: Evaluation 77
4.1 Construct Metrics for CPP Domain 80
4.2 Query Execution Results 82
4.3 IDE Integration 86

4.4 Framework Evaluation for a Comment Analyser 89
4.5 Conclusion 91
Chapter 5: Conclusion and Future Work 94
5.1 Supporting Construct Relationships 96
5.2 Proposed Query Results Visualisation... 97
5.3 Framework Enhancements.. 104
5.4 Conclusion 107
Bibliography 109
Appendix.. 113

List of Tables

Table 1: Possible table view showing results of a query on LINUX flags. 22
Table 2: C-CLEAR tooling framework features. 28
Table 3: List of C-CLEAR Eclipse plug-ins. 38
Table 4: Total number of CPP flags and affected files. 81
Table 5: Breakdown of total CPP flags based on how many files contain each flag. 81
Table 6: Proposed behaviour of Query Results Visualisation. 102
Table 7: Token types defined for CPP directives. 113

List of Figures

Figure 1: Problem overview showing separation of integrated tooling and Domain 5
Figure 2: Semaphore macros in Harmony, *linux/thrdsup.h* 7
Figure 3: MAKAO Build dependency graph for Linux 2.4 [11] 10
Figure 4: Interactive Class diagram in IBM's RAD for Java elements in a source package. 14
Figure 5: Radial visualisation in SolidSX. 15
Figure 6: Components that must be defined by a domain. 27
Figure 7: High-level C-CLEAR architecture, including data flow. 30
Figure 8: Some of the CPP domain tokens. 35
Figure 9: Interface for a Syntax Model element. 41
Figure 10: Example used to construct a Syntax Model in Figure 11 42
Figure 11: Syntax Model section created for the example shown in Figure 10 43
Figure 12: Interface defining a language that C-CLEAR can parse. 44
Figure 13: A light-weight API that decides if a provider should be invoked by C-CLEAR. 44
Figure 14: Contribution for parsing CPP directives via an Eclipse plug-in. 44
Figure 15: Contribution for CPP Language Provider via C-CLEAR Core extension point. 46
Figure 16: Lexical Tokeniser interface. 48
Figure 17: Interface defining a Syntax Parser in C-CLEAR. 49
Figure 18: Interface that a Language Provider must implement in order to construct Syntax Model nodes and mark constructs for the C-CLEAR index. 50
Figure 19: CPP directives grammar rule *cSection ::= block cSection | condCompSection cSection | ε* 54
Figure 20: Simple implementation for CPP construct marking. 56
Figure 21: Initialisation of Syntax Model Generator components in the Parser Controller. 59
Figure 22: Executing the parsing request via the Parser Controller. Note that if initialisation fails, parsing is terminated. 60
Figure 23: C-CLEAR context menu action that invokes C-CLEAR on a project selection, in this case, Kaffe JVM. 61
Figure 24: C-CLEAR Constructs View showing indexed constructs for Kaffe JVM, which in this case are CPP directive flags. 63
Figure 25: C-CLEAR Query Results View showing the results of a CPP query for finding duplicate code blocks controlled by the selected CPP flags. 66
Figure 26: C-CLEAR query preference page that allows query parameters to be configured. 67
Figure 27: Interface that must be implemented by a domain and contributed by a Query Provider. 69

Figure 28: A query request is processed by the Query Manager via the `runQuery(..)` API. 69
Figure 29: Actual query execution using a Syntax Model visitor task that evaluates nodes of interest to the query. 71
Figure 30: CPP query for comparing conditionally compiled code blocks. 72
Figure 31: Graph displaying the CPP flag usage breakdown from Table 5. 82
Figure 32: Query execution on all flags for GCC. 84
Figure 33: Query execution on all flags for OpenBSD src/bin directory. 85
Figure 34: Query execution on all flags for the Kaffe JVM. 85
Figure 35: Query execution on all flags for CVS. 86
Figure 36: Navigation from a query result to the C editor for a file in GCC. 87
Figure 37: Navigation from a query result to the C editor for a file in OpenBSD. 88
Figure 38: Navigation from a query result to the C editor for a file in the Kaffe JVM. 88
Figure 39: Navigation from a query result to the C editor for a file in CVS. 89
Figure 40: Prototype visualisation for C-CLEAR. 99
Figure 41: Proposed visualisation where horizontal blocks indicate code areas affected by a configuration construct. 101
Figure 42: Proposed visualisation and interaction properties in a second-level file view. 103
Figure 43: Levenshtein analyser used by the CPP query. 114

Chapter 1: Introduction and Related Work

Software systems are frequently built for a variety of platforms and environments, and may be configurable at both the build script level as well as a fine grained source level. Moreover, as these software systems evolve to adapt to changing hardware and execution environments, their source code becomes more complex with support for new paradigms, platforms and requirements. In many cases, decoupling the source from its runtime environment may in fact not be feasible, requiring the source to be aware of the different runtime scenarios for which the system is being built. In systems where lines of code may reach hundreds of thousands or millions, determining which locations or components require changes to support new requirements may be a daunting task. The problem is further magnified if no design artefact or conceptual abstraction is present to aid a developer in understanding the software's architecture, intention, or intricate communication and dependencies between components.

Management and maintenance of this kind of large, highly configurable source code presents complexities that are typically addressed through tooling, some of which will be covered in subsequent sections. Tools may generate levels of abstraction that facilitate analysis using high level representations of the software that are meaningful to a user. Some tools are designed as standalone applications [11, 7]. Although often sufficient to address a particular software maintenance issue, standalone applications typically do not integrate into a development environment where they would be used as part of the development process.

On the other hand, newer tools are becoming integrated into popular Integrated Development Environments (IDE) like Eclipse [20], and may present different interactive abstractions of a source code in the same environment where development occurs. Ultimately, this may assist developers in directly maintaining code from higher level views.

Many of these tools typically leverage existing libraries and frameworks, particularly those that provide UI, I/O handling and visualisation. Additionally, IBM's RAD [28] further uses UML [55] metamodeling to model software systems, presenting a wide variety of Graphical Modeling Framework (GMF) [26] based tooling for Software Visualisation. If the metadata definition is not sufficient to accurately abstract a software system, the tooling framework may also define hooks or extension points where additional information about a code base can be provided through an Application Programming Interface (API) [9].

As integrated development tooling is commonly used in maintaining source and software systems, as it is evident in the popularity of the Eclipse IDE, as well as other development environments like NetBeans [43] and IntelliJ [29], designing and implementing an integrated tool that assists developers in analysing concerns of interest in the source merits formal studies, in particular if the tools are reusable and extensible, and may more readily adapt to changes in the software properties like implementation language, build mechanisms, and version control. Among other things, extensible tooling frameworks reduces the issue of having separate, disjoint tools that work independently to address different aspects of a software system, saves time in terms of tooling development and adaptation to future changes, where older tooling may be rendered

obsolete by changes in the aforementioned software system properties. The design and implementation of an integrated, reusable, and extensible tooling framework for source analysis will be further addressed in subsequent chapters.

Reusable, integrated tools can be designed and implemented in different ways, from defining a common metadata schema that is used to automatically generate tooling, to developing a framework where most of the functionality is common and additional information about a software system is provided through an API and hooks. Another method is to extend existing functionality and couple it with specialised logic related to the software system being analysed.

This thesis presents a study of a framework based approach in the context of software configuration analysis, and potentially applicable to other areas such as low level language maintenance - much of the tool functionality is reusable and a common mechanism is implemented that is capable of generating and querying conceptual abstractions independent of the system being analysed. Specialised information about a system is still required. For example, definitions for the programming language used to implement the system. However, this specialised information is not coupled into the actual functionality. Rather, it is provided into the framework using hooks, like XML definitions used in Eclipse extension points, as well as API, and modeled into a generic structure that is understood by the common components of the tooling framework. This decouples the common framework from any specialised analysis tool built on top of it.

This thesis is by no means a comprehensive study of different tooling designs. Rather, it is a focused look at tooling frameworks where functionality can be decoupled into common, reusable mechanisms, and API is used to obtain properties and qualities

specific to a software system. As newer tools are designed and implemented as part of an IDE, focus will also be placed in developing a tooling framework where integration into a development environment is crucial.

Two key features of a tooling framework explored are:

1. Definition and separation of tooling functionality into common, extensible and reusable components from software system definitions using an API and clearly defined hooks. Evaluation of the extensible and reusable framework is focused on software configuration issues, including fluid conceptualisation of a software system into meaningful abstractions of interest to a user.
2. Automatic integration into an existing development environment.

The flow diagram in Figure 1 highlights the separation of concern that will be studied. The red area on the left side indicates decoupled, reusable tooling functionality, while the blue area on the right shows definitions and properties of a software system needed by the tooling side to properly extract and query data for a given software system. The data flow between the two areas is accomplished by the API defined by the tooling framework side, and implemented by the software system side, denoted as a Domain. The implemented API is then contributed into hooks defined by the framework.

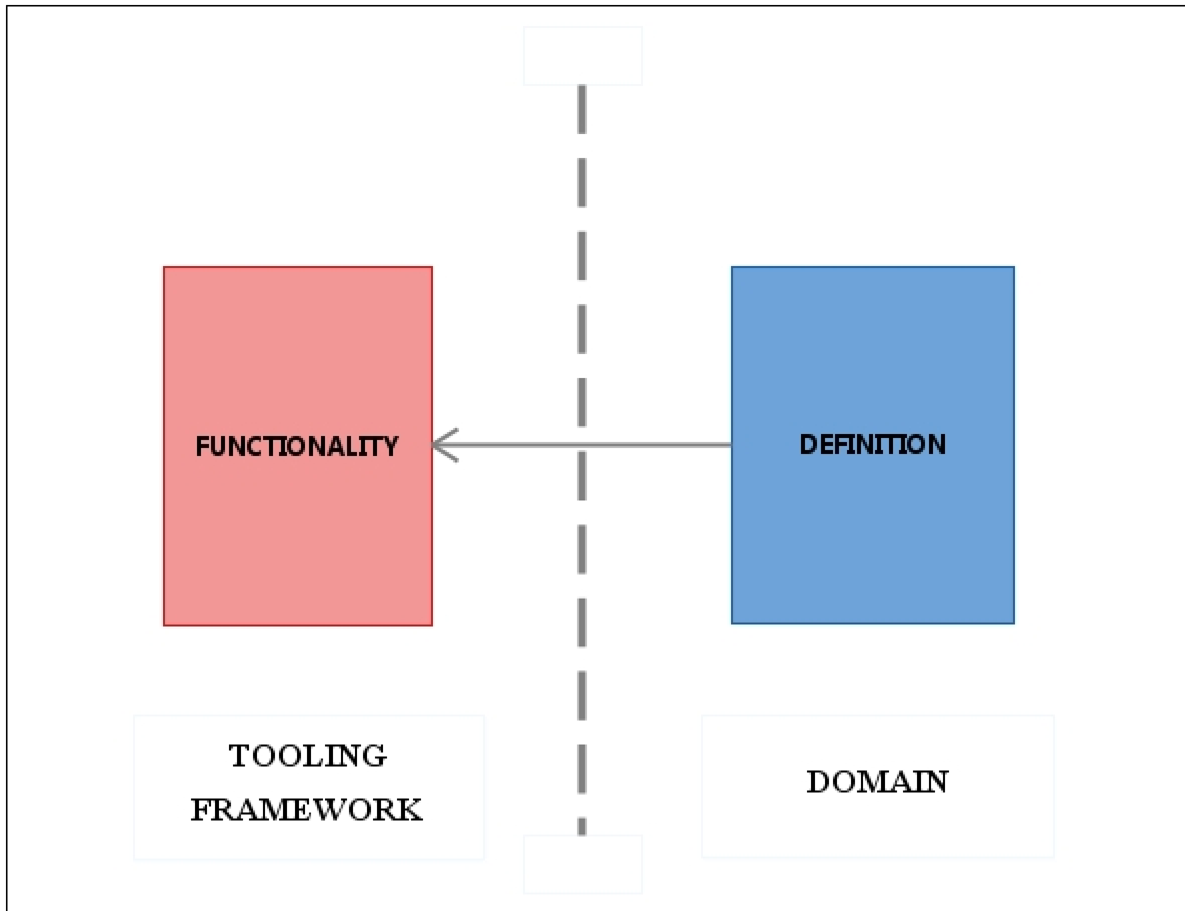

Figure 1: Problem overview showing separation of integrated tooling and Domain

1.1 Software Configuration Problems

To present motivational groundwork for developing integrated, reusable tooling, three main problems currently encountered in software maintenance and evolution of complex, configurable code bases will be described. In order to ensure a representative set of examples of system software, the main focus will be on Java Virtual Machines (JVM), parallel support in multi-core environments, and build script configuration and maintenance.

It is not uncommon for modern software systems like JVMs to lack portability and require extensive source level variation for execution in particular environments. For

instance, JVMs like Apache Harmony [8] and Kaffe [33] require subtle divergences based on whether the Java runtime is executing on a Linux or Windows operating system [44, 45]. VM architectures are also highly coupled with poor abstraction for subsystems, making it hard to infer functionality without also considering interactions with other components [39], resulting in slight divergences due to the configuration (typically hidden in build files), which is often harder to interpret based on configuration options alone, without looking at other dependencies.

Another area where software systems are rapidly evolving is in response to multi-core programming, which has shown that traditional sequential logic may require the introduction of parallel programming API to leverage additional processing power [14]. Changes in hardware, like moving to a multi-core environment may require alterations to sequential source code where parallelization support is implemented using popular API like Message Passing Interface (MPI) [40] or Open Multi-Processing (OpenMP) [48].

In both of these cases, where changes reflect differences in operating systems or the adaptation of new programming paradigms leveraging hardware like multi-core systems, variations can be widespread—touching not only on various locations in the same file, but affecting many files across the system. In addition, the technology used to inline variations at the source level may result in reduced separation between common code and platform variations thereby lowering the reusability of components. In particular, Harmony and Kaffe are written in C and include heavy use of C pre-processor (CPP) directives to implement variations at the source level. A comprehensive study of the complexities introduced by CPP directives across a wide variety of code bases has

shown that 8.4% of programming lines are made up of CPP directives, in addition to 25% that expand macros as well as 37% that are conditionally included using `#if` [21].

An example of a system that relies heavily on source level variation is Harmony, an open source JVM that uses CPP to redefine macros for semaphores in its threading support. Figure 2 shows the semaphore macro defined with thread memory allocation functionality if the `LINUX` flag is set, or an empty definition for all other cases, including Windows. The developer therefore has to cope with alternative implementations for particular builds, and may prove a daunting maintenance task for large software like operating systems and JVMs.

```
/* SEM_CREATE */
#if defined(LINUX)
#define SEM_CREATE(initValue)
thread_malloc(NULL, sizeof(OSSEMAPHORE))
#else
#define SEM_CREATE(initValue)
#endif
/* SEM_INIT */
#if defined(LINUX)
#define SEM_INIT (sm, pshrd, inval)
(sem_init((sem_t*)sm, pshrd, inval))
#else
#define SEM_INIT(sm,pshrd,inval)
#endif
```

Figure 2: Semaphore macros in Harmony, *linux/thrdsup.h*

Truly parallel programming will further introduce new paradigms and complexities that will affect sequential systems when ported to multi-core architectures [32]. For example, additional considerations need to be taken into account when dealing with synchronisation. Traditional schemes that avoid deadlock are difficult to reason about with respect to composition and therefore problematic with respect to parallel programming. Furthermore, typical spin-lock or interrupt-based schemes are either wasteful in terms of processor utilization or have high overhead. Alternate solutions for synchronisation that may prove to be more suitable to parallel programming include

transactional memory [12] and message passing [32]. Parallel programming may be associated with certain types of configuration, where configuration options determine whether parallelisation should be used instead of sequential logic. As with other configuration, parallel support may be highly scattered and hard to comprehend by looking at source code alone.

Another problem that highlights configuration issues in software evolution is in the maintenance of build scripts. Research on the Linux kernel has shown that source code changes do not occur in isolation, but rather require equivalent changes in the build system as new modules are added or existing ones moved in order for the system to compile [10, 11]. Additional studies have shown that although software systems may exhibit functional modularity, reusability is lost at the build level as seen with Graphviz , an open source graph tool, where circular dependencies and modular components that are unavailable for external use result in external software using this tool to have unnecessary dependencies [38].

Many tools exist that address each of these problems [11, 7, 1, 45, 52]. Some of these tools are standalone applications, while others integrate into an IDE. An IDE typically includes source file management, version control, compilation and debugging, and editing through source editors. These tools provide varying levels of abstraction and conceptual extraction to facilitate the maintenance of the software.

1.2 Related Tooling

A look at existing tools that address various software maintenance issues provides a basis for designing a tooling framework for software configuration analysis. The two areas that will be studied are Software Build Systems, as it is one of the issues covered in

the previous section, and Historical Analysis tools. In addition, data abstraction and visualisation will be covered in greater depth and presented will be presented as a valuable enhancement to software analysis tools, where abstraction may aid in software comprehension, as well as serve for meaningful context in which to conduct analysis queries.

1.2.1 Software Build Systems

A multitude of tools exist that provide a variety of mechanisms to analyse and maintain complex code bases that have undergone extensive evolutionary changes, and are used to address some of the issues mentioned in Section 1.1. They range from prototypes that perform specific functionality to extensible, integrated tools with strong emphasis on Software Visualisation [1, 52].

MAKAO is a visualisation tool that allows build system analysis via a build dependency graph, where nodes indicate files, and are colour-coded based on file type, edges show dependencies, and clusters of nodes are enclosed in a convex hull to indicate a particular build script [11]. In addition to visualisation, the tool allows users to navigate and edit script files, as well as query both static and dynamic information about build scripts, including build script variable values. Figure 3 shows visualisation of the build dependencies for Linux 2.4 [11].

MAKAO shows two important aspects required in a tool: the ability to interact with the displayed data, and a mechanism to query the visualised data. Both of these features should ideally form part of an integrated tooling framework, and furthermore be reusable as to allow different tools built on top of the framework to leverage the same functionality, and will be covered in greater detail in Chapter 2.

In addition, Figure 3 shows how certain tools employ visualisation that may render configuration issues in a graphical form, but are too complex and cluttered for comprehension. This layer of abstraction does not provide enhanced cognition of configuration issues, yet several tools like MAKAO employ visualisation that does not necessarily aid in the analysis and maintenance of configurable software systems. Chapter 2 proposes a more suitable visualisation of configuration using tables that may also be potentially applicable to other analysis areas.

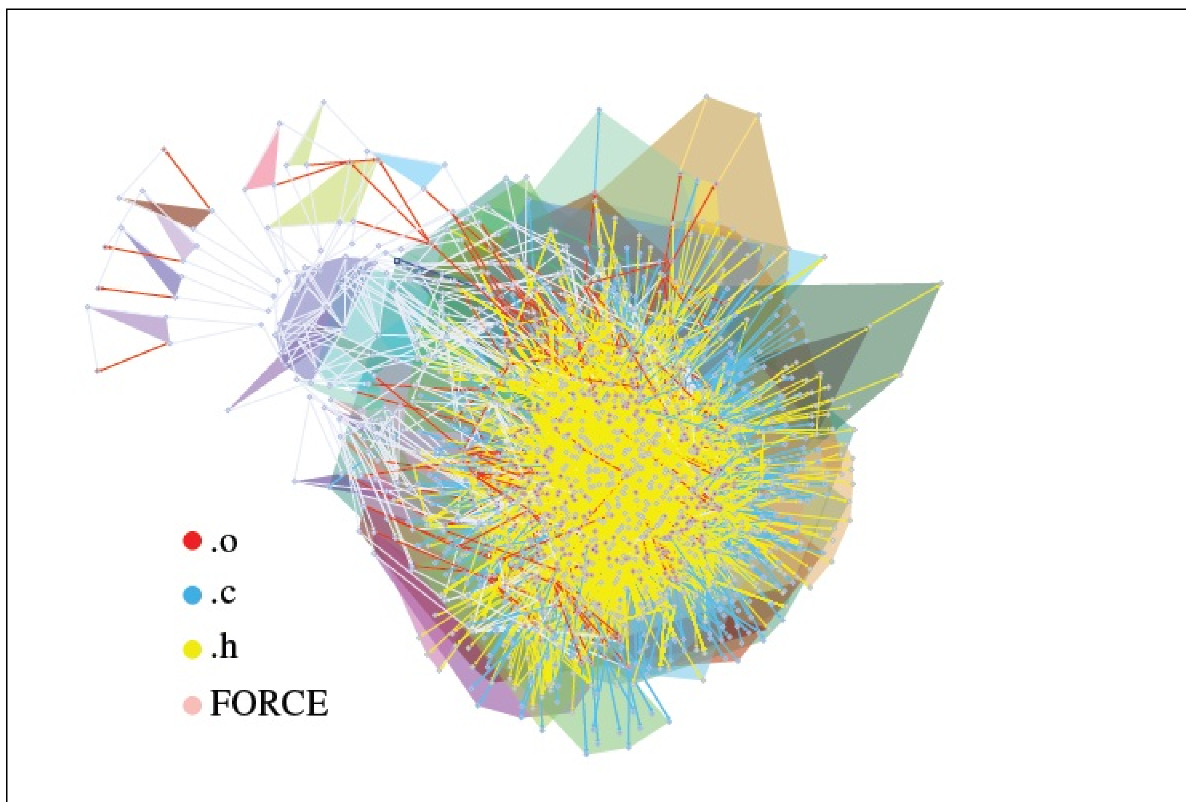

Figure 3: MAKAO Build dependency graph for Linux 2.4 [11]

The one drawback, however, is that MAKAO does not integrate with commonly used IDEs like Eclipse, and therefore developers that utilise Eclipse for C development and build management do not benefit outright from integrated visualisation of build

dependency. Ideally, changes in the build system through Eclipse would automatically reflect visualisation deltas in a hypothetical MAKAO Eclipse plug-in.

1.2.2 Historical Analysis

Historical analysis across different versions of a software system is also an area of focus for developing software maintenance tooling. As they are based on historical analysis of software evolution, part of the tooling functionality is the ability to adjust analysis parameters to adjust the scope of different software versions being analysed. However they do not present a common, integrated analysis mechanism that may be reusable across the different tools. This thesis will present a possible, reusable mechanism for adjusting analysis mechanisms in an integrated development environment.

Motive is a tool that visualises the effect of change sets in Java based systems under CVS control [7]. Like MAKAO it allows for queries to be performed on the visualised data, and in addition, has a configurable slider that adjusts the time period for which modification records for a particular system are to be analysed.

Hassan et al. [5] propose a Source Sticky Notes mechanism in a Software Reflexion Framework. In this work, dependency gaps between proposed concepts based on acquired knowledge of a software system and extracted architecture using automated tooling are bridged by attributes for each dependency, such as date of modification and modification comments.

Foo et al. [35] present an automated solution in Java that measures performance changes in regression testing by using a repository of performance regression testing, and propose a possible solution for handling evolutionary changes that would impact

regression testing performance. The solution indicates the possible use of a slider to select tests that better suite a more recent version of a software system.

The ability to adjust certain properties that narrow or expand the scope of extracted data or query results, either through sliders as shown in previous work, or through numeric values via a preferences User Interface (UI) as described in Chapter 2, should be a reusable component of a tooling framework that is geared towards software analysis. This is particularly true in cases where visualised data needs to be scaled in order to aid understanding of issues that may be of interest to a developer.

Another tool to look at is C-REX [3, 4], a source code extractor which uses a dependency change analysis algorithm, where facts about a software system are obtained from different versions, and the authors state the importance of empirical analysis in software evolution.

1.2.3 Abstraction and Visualisation

Abstraction may be a useful component of software maintenance tool, particularly when it comes to comprehension and querying of large, complex software systems. Prior research with tools like BEAGLE [49] has shown the need for source based analysis. However, software concerns like configuration code for specific runtime environments may be highly scattered across many files, and concerns may not be easily conceptualised by viewing source code alone, therefore source analysis tools may offer higher level views of a software system that accurately map to the source code, allowing developers to better understand functionality and identify components in a code base, as well as in certain cases maintain the code directly from the visualisation, as in the case of IBM's Rational Application Developer (RAD) [28].

Rajlich et al. [56] identified the need for abstracting concepts and determining concept location in order to effectively make changes to a software system. Further research has shown that source level pattern matching and code browsing can only provide limited abstraction as the usefulness of such methodologies decreases as software systems become larger and more complex [51]. A better approach is presented by the FEAT tool [42], where concern graphs are shown to be more effective in analysing Java programmes than just observing lines of code.

However, sometimes concerns cannot be directly parsed from language definitions of a code base without further input from a particular software system. For example, research has shown that parsing C code with heavy CPP usage often results in incomplete or inaccurate syntax trees if the code is parsed before pre-processing [9, 18]. Alternatively, syntactically correct C models can be generated after pre-processing, but only for one particular configuration. This will not assist developers in extracting meaningful concept graphs across different configurations. Instead, the research showed that using API while parsing a software system can more accurately identify concerns than just straightforward language parsing [9]. The use of an API to obtain concepts from a software system will be explored further in Chapters 2 and 3.

Some tools such as SPOOL [50] present an interactive visualisation of source code and higher level source code models through pattern-based design recovery. IBM's RAD allows Java elements in a project to be visualised in UML-based diagrams, as seen in Figure 4, where a diagram showing Java relationships for a class called `SyntaxParserConfig` allows members of a class to be re-factored directly from the diagram. In this case, the `resetErrorDetection()` method can be renamed right

from the `CCLRSyntaxParser` class view as opposed to a source code editor. Other tools like SolidSX [52] employ radial visualisation where source code elements like methods and fields are found along a ring and colour-coded by type, and relationships between these elements are bundled into connectors that traverse the centre of the ring, shown in Figure 5. However, as with the MAKAO visualisation in Figure 3, the abstraction in Figure 5 may prove to be too complex to improve comprehension of fine grained configuration concerns. Chapter 2 will propose a common mechanism that allows the extraction of higher level concepts that can be used for improved comprehension and analysis context independent of the software system that is being analysed and maintained, although emphasis will be on the mechanism rather than visualisation.

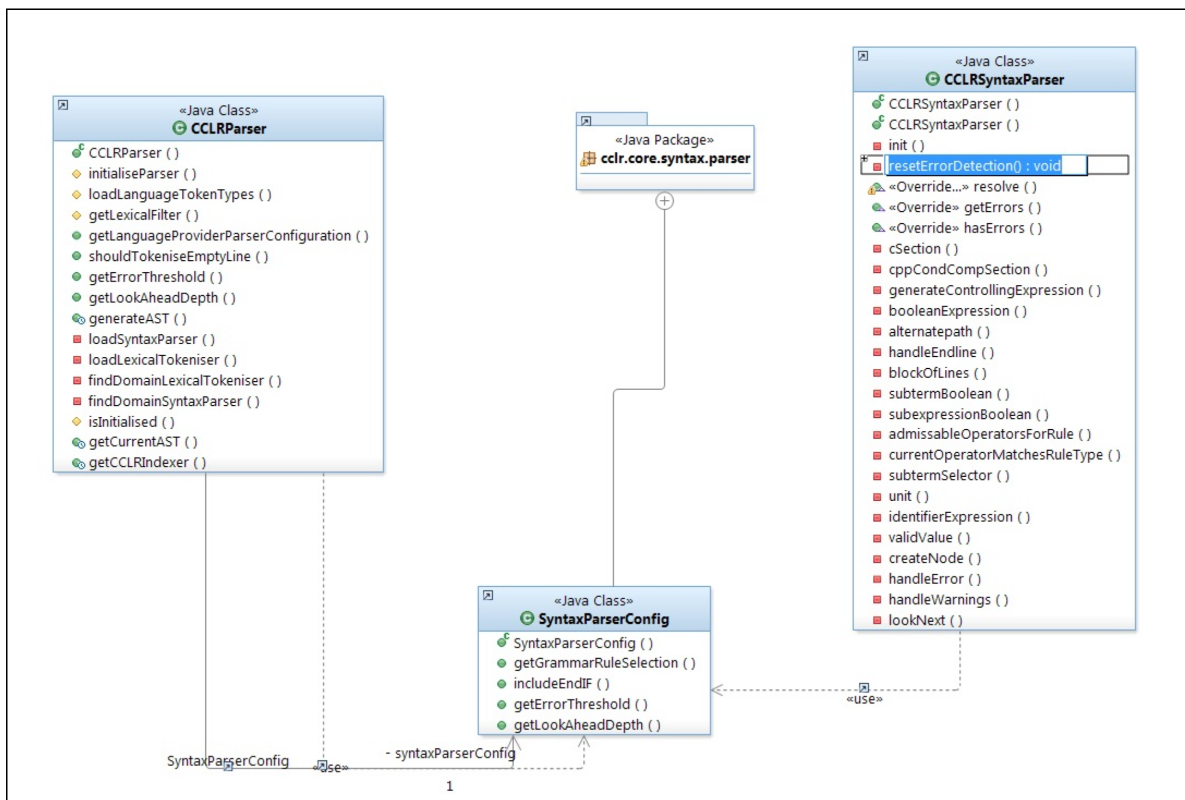

Figure 4: Interactive Class diagram in IBM’s RAD for Java elements in a source package.

Figure 5: Radial visualisation in SolidSX.

1.3 Need for Tools for Software Analysis

Tools are designed to address a variety of different software maintenance issues that sometimes cannot be readily be observed by simple source code viewing, and furthermore reduce time and resources in maintaining software. Code stability is one of the areas where tooling aids developers in gauging the overall health of a software

system, and historical change analysis is one of the areas used to detect instability. Tools used to examine frequent changes in a code component indicate instability and therefore low implementation quality [34]. Furthermore, metamodels like Hismo [53] can be used to detect whether so-called GodClasses, which are classes that centralise intelligence, are responsible for code instability due to frequent changes. GodClasses are considered harmless if the tool shows low historic changes. Tools that deal with version changes can also be used to detect dependencies that may result in non-compilation if a particular patch or version of that software is not applied before another more recent version [27].

Tools like MAKAO can also mitigate development slowdown, as without build script based tooling, relationships between source code and build scripts cannot be determined accurately, and the impact of making changes at source level like moving a file may result in the adoption of a new build system [11].

Other software maintenance issues that can be analysed through tooling are types of structural changes that a software system undergoes in a given period of time. Alam et al. [46] showed that 50% of all changes in a quarter for a software system are built-on-new changes, meaning changes built on fresh structures. Such information can aid development managers to assess progress during development iterations.

In addition, tools help in the detection of evolutionary patterns which indicate software volatility over its life cycle. Barry et al. [19] determined four groups of software systems, ranging from code bases that exhibited low volatility most of the time, to systems that started low but attained high volatility, to finally those that stayed volatile all the time.

Studies have shown that the use of software maintenance tools can reduce change efforts across families of products or product lines, where tooling decreases the need to reapply design patterns across different version of the same software system. In particular, one study showed that a software maintenance tool reduced change effort by as much as 40% [17].

Linux has been a major target of studies due to its popular use and long development history. Architectural analysis of Linux as shown that automated tools are beneficial in extracting architecture, and furthermore show a disparity between manually derived high level conceptual graphs and actual architecture, where the high level presentation did not take into account subsystem dependencies [30].

Although these examples have shown the benefits of using tooling to detect software design issues and reduce time spent in maintenance, many of these tools are disjoint, do not integrate into a common development environment, and do not have reusable, extensible components where adaptation to changes in software properties are readily handled. The efficiency of a tool in reducing software maintenance costs may be lost over time if the tool does not adapt to changes in software properties. This thesis postulates that a common, extensible, and reusable tooling framework can potentially improve the long term usefulness of tools, and maintain the tools efficiency in reducing maintenance time.

1.4 Common Tooling Features

Various studies have shown that software maintenance tools should possess certain qualities to make them useful to developers. Keller et al. [50] identified that certain interacting qualities are important in a software analysis and maintenance tool,

and include clear human understanding of all tool components, automatic functionality, visual representation and flexible visual transformations. Other research has shown that IDE integration is the primary feature sought after by tool users [1]. Additional features include scalability, ease of use, quick learning, robustness, and having similar look and feel to other tools and the IDE itself. However, lack of IDE integration was listed as the main factor in limiting tool usage, particularly in industry.

Telea et al. [2] also lists several key features necessary for a fact generating tool:

1. Generation of high level facts like call graphs
2. Generation of low level facts like syntax trees
3. Mechanism to select subsets of extracted data of interest to a user
4. Ability to query and analyse the subsets and calculate quality and metrics
5. Provision of an interactive mechanism to understand and navigate the data
6. Integration between all features

Additional user experience studies on several software maintenance tools have shown that a combination of comprehension strategies, ease of accessing and switching between strategies, and reduced overhead for understanding a code base are desirable qualities in a tool [41].

The tooling framework proposed for this thesis will examine some of these features. Conscious effort will be placed on separating features that fall under tooling functionality into a tooling framework, and software related properties and definitions into a Domain side, which are then contributed into the tooling framework through well-defined hooks. For subsequent sections and chapters, a Domain is defined as a set of related characteristics of a software system, such as the implementation programming language (e.g., C or Java), or a mechanism used to handle conditional compilation and

configuration like CPP. Complex software systems can therefore have various domains associated with them, with each focusing on a particular aspect of the system.

Some of the various ways of designing a software configuration analysis tool include defining metadata schemes to automatically generate tooling, leveraging existing frameworks like IBM's RAD, and developing a tool that works on intermediate object representations of a code, like Eisenbarth's C concept analysis tool [54]. The thesis focuses on an API based approach for information transfer between a domain and the functional tooling side.

Two reasons to study an API-based mechanism are:

1. An API may allow for tools built on top of a tooling framework to identify concepts in the code that are not readily defined by the implementation language during the actual parsing of a source code using an existing common concept marking mechanism. These concepts may aid software comprehension, and furthermore serve as context for analysis.
2. Since common functionality is decoupled from software domain definition, an API-based approach can determine what type of information is needed from a software Domain in order for the tooling framework to extract and analyse data, and at what points in the functional flow is the information needed.

1.5 Use Case Study

Eclipse has already shown the power of using a common, extensible framework for tool development, where various tools can collectively present different views of the same source project [24, 23, 16]. Eclipse's powerful extensibility and UI support allows for different types of tools to be integrated into the same environment in which development is performed. Therefore, we use Eclipse as our base for a tooling framework that can address issues dealing with software configuration, and demonstrate how this framework would ideally integrate into Eclipse.

To showcase the need for Eclipse-based tooling, consider the scenario of a software engineer maintaining a fictional Java Virtual Machine called **TestJVM** and would like to find and analyse code patterns that may be commonly used across different configurations. The software engineer also uses Eclipse's C Development Tooling (CDT) [13] for source development, and would prefer that any code patterns found can be visually mapped to the configurations selected by the engineer. Furthermore, the engineer would like the analysis results to be integrated into CDT, such that navigation to source locations using CDT editors is possible.

The software engineer would like to find all locations in the JVM source that are conditionally compiled for Linux and determine what code patterns are found more than once for this configuration, and in which locations. Eclipse already provides a robust search mechanism that can find patterns and even C constructs, but as the engineer does not know what code patterns to look for, only the configuration that needs to be analysed, Eclipse search mechanism can at best search for all areas where the `LINUX` or related variations of this CPP flag is used. This may result in vast amount of information that may be hard to sift through in Eclipse's search view. Instead, the software engineer would like implement a mechanism that is more specifically geared towards analysing syntactically similar conditionally compiled code for particular build configurations, and be able to view all available configurations, and potentially reuse the tool to discover and display other available configurations aside from Linux. Therefore the engineer would like to implement a tool that is more specialised towards discovering and displaying all CPP-based configurations, and querying conditionally compiled code. Finally, as the analysis criteria may change in the future, where requirements for searching for

syntactically similar conditionally compiled code may change to some other goal, the engineer would like to reuse the tooling infrastructure as much as possible for future tool development, and just focus on implementing future analysis queries, rather than have to worry about tooling functionality like source parsing, UI and visualisation of analysis results.

As a tool developer, the engineer would need to parse CPP directives and analyse those sections of conditionally compiled code that would be pre-processed if the `LINUX` flags are set in the build process. Building a tool from scratch that would parse CPP directives into a model and integrate into Eclipse may prove to be a major task, in particular if it has to be revised or re-implemented to accommodate for future requirements.

Instead, this thesis proposes that the engineer could use an existing tooling framework that already contains parsing mechanisms, as well as ways of marking constructs in the code that may be meaningful to the engineer. Furthermore, the reusable construct marking functionality should be automatically integrated into the framework's querying capabilities and UI, as to allow a tool user to select whatever constructs engineer decides are of interest for further analysis. In this case, those constructs would be CPP configuration flags like `LINUX`. Furthermore, all the UI and IDE integration with other components like source editors and file management would be fully leveraged by the engineer. Specifically, we propose an extensible framework that simply allows the engineer to define: (1) a set of tokens to parse (in this case CPP directives), (2) language rules to generate a model based on CPP syntax, (3) data that indicates configuration, and (4) query operations that perform analysis on C pre-processed code blocks based on a

selection of flags in an existing tooling UI. This would save development time and allow the engineer to focus on aspects of the tool that pertain specifically to the Domain the engineer is analysing and maintaining. No UI coding would be required, and integration with the IDE is automatic such that navigation from an analysis result automatically opens the correct IDE source editor and highlights the actual location of the result.

Table 1 shows the results that the engineer may be looking for. Ideally this data would be a table tree viewer, where each row expands into a tree. At the root level, the Affected Files column would show the total number of files containing the result. Expanding the row will show all the file names and line numbers for each result, and double clicking on any one of them would open the Eclipse CDT editor at that location, allowing the user to modify the code.

Flag	Source	Affected Files
`LINUX64/LINUX32`	`if(buffered()) {..}`	12
`LINUX32`	`processBuffer();`	20

Table 1: Possible table view showing results of a query on LINUX flags.

The four components that the engineer would have to contribute through the extensible tooling framework are:

1. A list of tokens to parse.
2. Language rules that would build a model structure that represents a conditionally compiled source. The engineer would not have to define its own model and worry about integrating it into the framework, but rather use an existing set of API that define the structure elements of the model, and simply use the API to build the structure in a way that conforms to the configuration source being parsed and analysed.
3. CPP directive flags like `LINUX32` and `LINUX64` which may conceptually represent a Linux configuration, and can serve as an analysis context for launching queries. The engineer would not have to implement

any integration with the framework telling it that these constructs should be interpreted as CPP flags, and displayed in a UI view. Rather, the engineer would only have to implement an API that marks certain parsed data as selectable constructs of interest for further analysis purposes, and leave the management and display of these constructs to the framework.

4. A query that performs analysis on model elements built by the language rules and which pertain to selected CPP flags from the framework's common UI view. The execution of the query and management and display of query results will all be taken care of by the framework.

This way, the engineer would rely on the tooling framework's common, reusable UI and functionality to do the following:

1. Start the parsing and modeling process via a context menu action when the source project is selected in Eclipse.
2. Display all the CPP flags as constructs that may conceptually represent a build configuration and would narrow the focus of a particular query performed on conditionally compiled code controlled by those flags. In this case, the view should allow the user to select the `LINUX32` and `LINUX64` flags.
3. Execute a query through the framework UI and display the results in a framework view similar to the table shown in Table 1.
4. Select a result and navigate to the location in the same Eclipse C editor used for development.
5. Adjust parameters of the query using an existing UI, allowing the analysis scope to be narrowed or expanded. For example, the engineer would have the ability to find conditionally compiled patterns for `LINUX` builds that are found at least 3 times simply by entering a number 3 in some generic UI control for the framework's common query mechanism preferences.

This scenario shows the need for an integrated tooling framework that minimises tool-specific implementation, in particular avoiding any UI development, data and model management, or IDE integration. The tooling framework must have clearly defined extensibility points where an engineer can readily contribute specifics about the Domain the developer is working on. The framework should also provide automatic integration

with other commonly used components of the IDE, like the source file editor, compiler, build system and source project view.

This thesis proposes an Eclipse tooling framework called C-CLEAR which embodies all the aspects mentioned in this use case. Chapter 2 covers the design of C-CLEAR in greater detail, while Chapter 3 focuses on specific implementation of the tool. Chapter 4 presents an evaluation of C-CLEAR on various code bases, including the Kaffe JVM. Chapter 5 describes some limitations of the tool and presents future enhancements of C-CLEAR, particularly in the area of Software Visualisation.

Chapter 2: Design

C-CLEAR is our proposed prototype for studying the use of integrated tooling frameworks to address issues related to software analysis—in particular, source-level support for configurations. Our goal is to evaluate an approach to build common framework architecture and mechanisms that a tool developer can reuse for software analysis that also integrates into an IDE. One of the principle contributions of C-CLEAR is the separation of common components that integrate into an existing IDE, such as Eclipse, from Domain specific software system definitions that are incorporated into the common components through extension points. This approach would allow tool developers to primarily focus on Domain specific implementation related to analysis, and even implement various types of tools that perform different operations and present different views of the same software system in a common IDE using the same C-CLEAR tooling framework.

Generally, the common, IDE-integrated components of the framework would include the UI, source-parsing, event handling between all the components, indexing of selected concepts or constructs in a source that can improve understanding of a software system, and source model definition. These are intended to be reused regardless of the source type being analysed. On the other hand, the software specific attributes and properties would be contained in a Domain definition, like for example keywords in a language as well as language syntax rules. Other Domain components would be API implementations that mark facts in a source as potential abstract concepts that can aid a tool user in better understanding and querying specific portions of a software system. The

Domain would also provide a query that would understand these abstract concepts, but would not be responsible for actually executing the query or handling communication between the query and other parts of the framework. The functional aspect of the query would fall under the control of the common framework.

Consequently, C-CLEAR is designed along a clear separation between a Domain and an integrated common framework that actually performs the tooling function. Communication between the two parts is attained through an API defined by the framework, implemented by the Domain, and contributed by the Domain into the framework using framework hooks. Leveraging the common framework, the tool developer would only be focused on implementing Domain logic, and rely on the framework for the functional side of the tool, including data extraction and modeling, UI and automatic integration into the development platform.

2.1 Design Specifics

C-CLEAR's purpose is to help developers build tools to analyse complex software configuration within an IDE using extracted data and abstract concepts from a software system that is being analysed and maintained. However, C-CLEAR is not meant to be a universal solution for all possible issues that can be encountered in software configuration analysis, but rather a prototype to explore elements of a common tooling framework that include data and concept abstraction, data presentation, and automatic integration into an IDE. For example, a tool built on top of C-CLEAR would only need to focus on telling the framework what aspects of a source to parse and conceptualise, and would rely on the framework to extract the data, present it to a user using common UI, and allow queries to be performed on the data and concepts.

To determine what information to parse, the Domain portion of the tool needs to tell the framework: (1) which patterns to parse in a source, (2) what rules to use to build a model based on the parsed patterns, and (3) which selectable values in the model represent abstract concepts of interest to the domain and can be used as context or references for queries. As general analysis of data involves asking questions about information, a fourth component contributed by the domain is a query that can be performed by the framework on selected data. Figure 6 summarises the four concrete components that must be provided by the domain in order to build an integrated analysis tool that leverages a common tooling framework.

1. Patterns to tokenise.
2. Rules that build a generic model understood by the common tooling framework.
3. Constructs in the model that represent abstract concepts and are displayed to a user and allow for model querying.
4. Set of queries to be performed on selected constructs defined in Component #3.

Figure 6: Components that must be defined by a domain.

The tooling framework would be responsible for defining hook points where the domain can contribute the components in Figure 6. The domain developer would not have to implement any functionality, IDE integration or UI as that would all be leveraged from the tooling framework.

Table 2 proposes a list of basic common functional, integrated components that the tooling framework would need to implement, grouped into four categories of integration: into an IDE, UI, Core, and Query. Each of the components in Table 2 will be covered in greater detail in subsequent sections.

Integration	1. Automatic integration into an existing an IDE like Eclipse, including communication between the tooling framework and the IDE.
UI	1. **Project Selection**: UI that allows a user to select a source directory or set of source projects for analysis from an existing IDE view. 2. **Constructs View**: UI that displays domain-defined selectable constructs (Component #3 in Figure 6), and contains controls for launching a query. This view should use the IDE's UI framework to maintain the same look and feel as the IDE. 3. **Query Results View**: UI that displays query results, and allows navigation to existing integrated components of the IDE, like source file editors and directory or project views. As with the Constructs View, it should have the same look and feel as the rest of the IDE. 4. **Query Preference Page**: UI that allows users to fine tune parameters that affect the scale and accuracy of the query results. The preferences should be integrated in the same location as other IDE preferences.
Core	1. **Parser Controller**: Handles events and communication between existing IDE UI and platform, C-CLEAR defined UI, and the back-end C-CLEAR Core layers. It is also responsible for initialising the parsing and Syntax Model generation process, where the Syntax Model is a model representation of the parsed data. 2. **Lexical Tokeniser**: Component that generates tokens from an input, regardless of input type. 3. **Syntax Parser**: Builds a Syntax Model based on domain grammar rules and tokens provided by the Lexical Tokeniser. 4. **Generic Construct Marking**: Defines a common mechanism to allow domains to mark constructs in the Syntax Model, and adds them to a tooling framework index structure. These constructs represent conceptual abstractions in the source code, and are displayed to the user to facilitate querying on specific portions of the software system. 5. **Domain Extension Point**: Defines a hook or extension point where a domain has to contribute *Token Definitions* and *Grammar Rules* for the Lexical Tokeniser and Syntax Parser, respectively.
Query	1. **Query Controller**: Handles events and communication between existing IDE UI and platform, C-CLEAR defined UI, C-CLEAR Core Layer, and the Query Layer. It also initialises the Query Manager when a request is made for a query operation. 2. **Query Manager**: Loads and runs a query defined by a domain and manages results obtained from the query. 3. **Domain Extension Point**: Defines a hook where a domain has to contribute a query for the Query Manager

Table 2: C-CLEAR tooling framework features.

Figure 7 shows a high-level design diagram of C-CLEAR, embodying most of the features described in Table 2, as well as primary data flow between each of the layers and components.

Figure 7 is divided into two main areas, the C-CLEAR tooling framework and IDE in red, and the domain shown in the middle in blue. The three data flows shown indicate the following:

1. **Red Flow**: Starting from the Project Selection component in the common IDE UI and ending in the Constructs View in the C-CLEAR portion of the UI, the flow shows the parsing of an input source, generation of a Syntax Model, which is an abstraction of the parsed data, via the Lexical Tokeniser and Syntax Parser in the Core layer, and the display of marked constructs in the Constructs View.
2. **Green Flow**: Beginning from the Constructs View and ending at the Query Results View via C-CLEAR's Query layer, this flow shows the selection of constructs by a user for querying, and the subsequent display of results.
3. **Blue Flow**: Shows the three points in the framework that require contributions from the domain:
    i. **Token Definitions**: Required for the Lexical Tokeniser in the Core layer.
    ii. **Grammar Rules**: Required by the Syntax Parser in the Core layer.
    iii. **Query**: Required by the Query layer in order to perform an operation on selected constructs in the Constructs View.

The framework portion shows three main areas: UI**,** Core and Query**,** each of which is subdivided into sections indicated in light red. Starting at the top, the UI layer is integrated into the IDE's UI, and has two main sections: (1) existing IDE views that contain UI for source project/directory selection and invocation of C-CLEAR through a Project Selection component, such as a context menu action, and (2) C-CLEAR views implemented for the tooling framework that display conceptual constructs in the Constructs View and navigable query results in the Query Results View. These two C-CLEAR views are integrated into the IDE's main workbench UI where development,

source control and building are also performed. For brevity, other UI components like IDE source editors, source control and build views, as well as C-CLEAR's query preference page have been omitted.

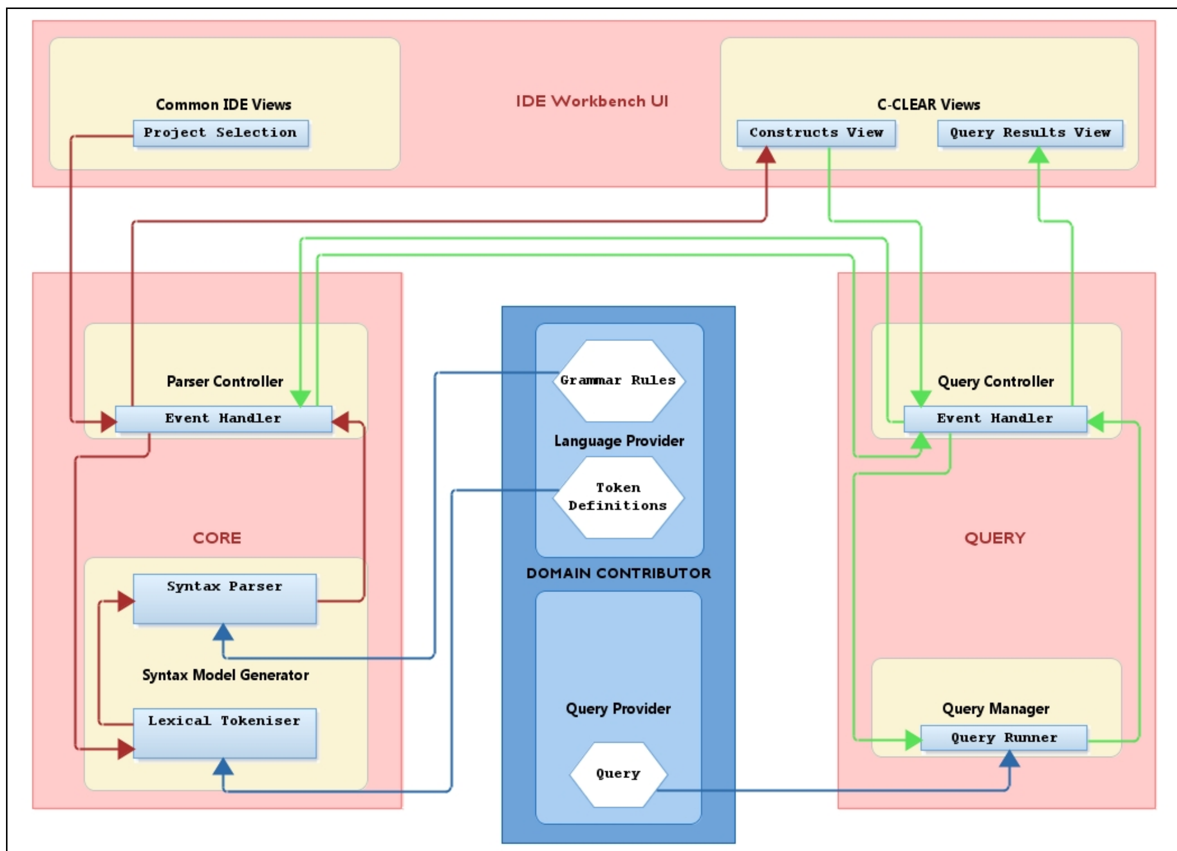

Figure 7: High-level C-CLEAR architecture, including data flow.

The Core layer has two main sections, the Parser Controller which handles events and communication between the lower Core layer and the UI, and the Syntax Model Generator layer which performs the parsing, modeling and construct marking of an input source through two stages:

1. *Lexical Tokeniser*: lowest component of C-CLEAR which performs parsing of an input into tokens based on Token Definitions provided by a domain.
2. *Syntax Parser*: next lowest component of the Core layer which builds a Syntax Model based on tokens fed by the Lexical Tokeniser and Grammar Rules provided by the domain. The Syntax Parser is also responsible for marking and indexing conceptual construct values (Component #3 in Figure 6).

The third and final component of the tooling framework is the Query layer, which has its own event and communication controller, and is responsible for processing a selection of constructs from the Construct View, obtaining the Syntax Model from the Core layer, executing a domain query on the selection and the Syntax Model through a Query Manager, and displaying the results in a Query Results View.

C-CLEAR has two main extension points or hooks where domain contributions are required. One is in the Core layer and the other in the Query section, as seen in Figure 7. Although the hooks are not specifically shown in the architecture diagram, they can be inferred by the two light blue areas in the central blue domain section. The domain must implement two providers, one for each hook, using an API defined by C-CLEAR's framework. The first provider, known as the Language Provider, contributes two domain components, Token Definitions and Grammar Rules, at two different stages of the Syntax Model generation process. The second provider, known as the Query Provider, contributes a query into the Query layer which only gets invoked during a Green flow.

The Token Definitions define language patterns to parse and tokenise, while the Grammar Rules performs two main functions: (1) building a model of the parsed data using syntax rules that define the language used in the software system, and (2) marking conceptual constructs in the parsed data while the model is being built. These constructs provide a higher level context for users to invoke queries, and are interpreted by a domain query during a query operation.

For the first contribution, as an input is being parsed by the Lexical Tokeniser in the Red flow, tokens are generated based on a list of token patterns provided by a Domain Language Provider and obtained through the Core's extension point. Upon receiving a set of tokens from the Lexical Tokeniser, the Syntax Parser requests Grammar Rules from the same Domain Language Provider through the same extension point in order to build a Syntax Model. The rules decide the structure of the Syntax Model, although the model elements themselves are generically defined by the framework.

The reason the model elements themselves are generic is because the Query and UI layers of the framework need to operate on a commonly understood model structure regardless of domain. While the Syntax Model is being generated, the Syntax Parser also requests the Language Provider to identify and mark elements in the model which represent higher level concepts and are then added to a generic index (not explicitly shown in the diagram). These elements are known as constructs, and provide an additional layer of abstraction that is intended to help users understand the domain code base and serve as a meaningful reference for query execution. These constructs are a selectable set of values that are displayed by the framework in the Constructs View once the parsing process and Syntax Model generation are complete. These same construct

values are used to perform queries on certain portions of the Syntax Model that map to those constructs. The mapping between the domain-marked constructs and the Syntax Model is handled by a generic index mechanism in the tooling framework. C-CLEAR's construct-based concept abstraction will be further explained through implementation details in Chapter 3.

The second hook in the Query layer allows a domain provider to perform operations on the Syntax Model based on constructs selected by a user in the Constructs View. Loading a query, executing it, and managing and displaying the query results are all handled by the C-CLEAR framework through the Query Manager, independent of the domain. C-CLEAR also handles the integration between the Query Results View and the rest of the IDE. In particular, C-CLEAR supports navigation to source files in the IDE's Project Selection View, as well as to locations in the file containing the query results, which are shown in the IDE's source editor.

The choice of what constructs to mark and the semantics of those constructs are determined by the domain in the Language and Query providers. To better understand C-CLEAR's design and functionality, a simplified description of the parsing and querying process will be explained based on the use case previously mentioned in Section 1.5.

2.2 Use Case Explored

Chapter 1 described a tool developer interested in finding Linux-configured C code to parallelise. That tool developer would then use C-CLEAR to implement a Language Provider for CPP directives, which would define a list of CPP directive tokens as seen in Figure 8 (a more complete list of all the CPP directive tokens is shown in the Appendix, Table 7). It is important to mention that although the list of CPP directives

handled by the provider is not complete, it is sufficient to evaluate the tool against different C sources, which is further covered in the evaluation provided in Chapter 4.

The CPP directives are then tokenised by the Lexical Tokeniser based on the CPP Token Definitions and then fed into the Syntax Parser which then asks the CPP language provider to match the tokens against a grammar rule. If a grammar rule is matched, for instance, a grammar rule that defines conditional compilation through `#if…#endif`, the Language Provider: (1) creates a Syntax Model element for the matched token, (2) sets a value for the element, which would be a CPP flag like `LINUX32` and `LINUX64,` and (3) marks these flags as conceptual constructs. C-CLEAR then adds these constructs to a generic index and maps them to sections of the Syntax Model that contain the constructs. The tooling framework also displays these constructs in the Constructs View, allowing a user to select either or both flags for further querying.

It is important to note that C-CLEAR is not aware that these constructs are CPP directive flags, or that they have any particular meaning to the domain, as there are no semantics about any domain hardwired into the tooling framework. The tooling framework only knows that certain facts of the Syntax Model have been marked by a particular Domain Language Provider and should be displayed in the Constructs View, but has no additional semantic knowledge. It is up to the domain query to interpret these constructs during a query operation.

The construct selection from the Constructs View are then used by C-CLEAR to execute a domain query on portions of the Syntax Model that correspond to that selection. For example, a possible CPP query would be to determine if the Syntax Model elements that map to `LINUX32` and `LINUX64` have conditional CPP directives like `#if`

that evaluate to true when the flags are selected. If so, the query would proceed to analyse the controlled C block to see if it matches code of interest to the developer, for instance, logic that handles larger buffer processing. In the simplest case, the query can use regular expressions on the controlled C block to check for patterns of interest to the developer. In more complicated cases, the query can rely on the IDE's compiled Abstract Syntax Tree and symbol table to perform query operations. For the purpose of evaluating C-CLEAR, a specific CPP query was implemented that uses a Levenshtein algorithm [6] to search for syntactically similar conditionally compiled code blocks, regardless of whether they are syntactically correct C code. This query is covered in more detail in Chapter 3.

If query conditions are met, the Syntax Model element is marked as a query result, and its properties and values are shown in the Query Results View. The values of the model elements include parsed data, such as the raw conditionally compiled C code in the case of a CPP domain, along with the following general properties that are domain independent: file name and starting line number in the source code containing the result, and the number of similar results found in the parsed code base.

```
IFDEF = "ifdef";
IFNDEF = "ifndef";
ELIF = "elif";
ELSE = "else";
ENDIF = "endif";
PDEFINE = "define";
INCLUDE = "include";
UNDEF = "undef";
PRAGMA = "pragma";
```

Figure 8: Some of the CPP domain tokens.

C-CLEAR's generic construct marking and indexing component is intended to show that a common mechanism can be used to abstract certain facts about the data that

permit higher level understanding of a source and allow more fine tuned querying to be performed on the data model. The previous example described a simple query that searches for conditionally compiled C code when the `LINUX32` and `LINUX64` flags are enabled, and which contains large buffer processing logic. In the actual C-CLEAR prototype we have further implemented a CPP domain which defines a query for searching possible redundancy in conditionally compiled code sections. Specifically, we wanted to provide support to verify if any similar code blocks exist that may merit modularisation, in particular if the logic may exhibit frequent changes.

The CPP query prototype essentially investigates code quality and conditional compilation usage using abstracted concepts such as CPP flags. It also facilitates refactoring duplicate configuration code into modules, at least for syntactically correct C code, although the CPP domain Contributor is only meant to be an evaluation test case and not a complete solution that handles any conditionally compiled code. Implementation of the CPP Language and Query Providers is covered in greater length in Chapter 3, along with a more detailed explanation of C-CLEAR's framework.

2.3 Summary

The design of C-CLEAR follows some of the expected features desired in a software analysis tool, particularly one that handles source level configuration. Chapter 1 presented two main goals of a tooling framework study based on domain and tooling functionality separation through well defined API: (1) establishing common integrated components that extract data and mark abstract concepts regardless of domain, and (2) integrating the tooling framework into an IDE.

C-CLEAR achieves a separation of tooling functionality from domain definition by establishing a set of components on the domain side which include Token Definitions, Grammar Rules and queries, and a series of common functional steps on the framework side. The framework steps include lexical tokenisation, syntax parsing, construct marking and indexing, UI data presentation, query execution, and query result display. Domain components are contributed into certain locations in the framework through framework hooks, and the contributions come in the form of concrete domain implementations of an API defined by the framework.

Section 1.4 mentioned that IDE integration is the biggest feature expected by users [1]. C-CLEAR is therefore implemented as a framework that integrates into Eclipse, and maintains the same look and feel as the rest of the IDE. Although Chapter 1 mentions visualisation as a key focus of software analysis tooling, C-CLEAR only provides a basic Constructs View that presents abstracted concepts in a simple and selectable table structure. Chapter 3 will cover implementation of key components of C-CLEAR.

Chapter 3: Implementation

This chapter focuses on in depth implementation of certain key components of C-CLEAR's design as described in Chapter 2. C-CLEAR is an integrated tooling framework for source configuration analysis in existing IDEs, such as Eclipse. Common steps to parse input, generate a Syntax Model, mark constructs in the parsed data, and present them to a user for query operations are re-used, regardless of domain or source type. However, in order for these common steps to produce data of value to a user, more specialised input is required from domain contributors. In particular language definitions such as token types, grammar rules, abstract domain constructs, and queries are required.

Eclipse is a widely used Java-based IDE, and therefore a reasonable choice for implementing C-CLEAR. Much of this chapter will cover the Java implementation of C-CLEAR as a set of Eclipse plug-ins. Three plug-ins contain the framework portion of C-CLEAR, and one plug-in defines a CPP domain, containing both CPP Language and Query Providers, as well as helper classes, shown in Table 3

`com.cclear.framework.core`	C-CLEAR Core layer, including extension point definition for contributing Domain Language Providers.
`com.cclear.framework.ui`	C-CLEAR integration into Eclipse workbench UI, including C-CLEAR views, context menu actions and query preference page.
`com.cclear.framework.query`	Query framework, including extension point definition where Domain Query Providers are contributed.
`com.cclear.domain.cpp`	Implementation for CPP domain, including CPP Language and Query Providers.

Table 3: List of C-CLEAR Eclipse plug-ins.

Implementations for key design features proposed in Chapter 2 will be covered in three sections, presenting an overview of each of the main layers of C-CLEAR as shown in Figure 7: Core, UI, and Query. Section 3.1 will overview six main components of the Core layer:

1. The generic, domain-independent Syntax Model in Section 3.1.1
2. The Domain Language Provider and framework hook for contributing the provider in Section 3.1.2
3. Lexical Tokeniser in Section 3.1.3
4. Syntax Parser in Section 3.1.4
5. Construct marking and indexing in Section 3.1.6
6. The Parser Controller in Section 3.1.7

Section 3.2 covers the following four UI components of C-CLEAR:

1. Project Selection in Section 3.2.1
2. Constructs View in Section 3.2.2
3. Query Results View in Section 3.2.3
4. Query Preferences Page in Section 3.2.4

Finally, Section 3.3 presents the last layer of C-CLEAR, which is the Query level. The two sections, 3.3.1 and 3.3.2, cover both the framework portion of the Query layer, where the Query Manager handles the loading and execution of a domain Query, and the domain side, where a CPP domain implements a query for computing syntactically similar conditionally compiled sections of code.

3.1 Core

C-CLEAR's core layer is implemented in a `com.cclear.framework.core` plug-in.

It contains all the Core feature implementations described in Chapter 2, including:

1. The event controller that handles communication between the core, UI, query layers as well as the Eclipse platform
2. Initialisation of the Syntax Model Generator
3. All interfaces that a Domain Language Provider must implement, including extension point definition that allows Language Provider contribution
4. Lexical Tokeniser
5. Syntax Parser
6. Syntax Model definition
7. Construct marking mechanism and indexing

The main component in the Core layer that manages the parsing process and generates a model is the Syntax Model Generator shown in Figure 7. One of the key aspects of the Core is the definition of the Syntax Model, as it represents an abstraction of the parsed data independent of the input type.

3.1.1 Syntax Model

In order to be reusable, C-CLEAR defines a generic Syntax Model that represents data parsed from a given input source. The model has to be domain independent as it is intended to be used by the UI and Query layers regardless of the type of software system that is being analysed. Domain Contributors are responsible for building the model based on domain-specified language rules during the syntax parsing stage, but the actual model is defined by C-CLEAR framework interfaces.

```
public interface IASTNode {

    public void insertFirst(IASTNode node);

    public void insertAfter(IASTNode afterThisNode, IASTNode node);

    public void insertBefore(IASTNode node, IASTNode beforeThisNode);

    public void insert(IASTNode node);

    public IASTNode getFirstChild();

    public IASTNode getLastChild();

    public IASTNode getNextChildAfter(IASTNode node);

    public IASTNode getParent();

    public void setParent(IASTNode parent);

    public INodeValue getNodeValue();

    public INodeType getNodeType();
}
```

Figure 9: Interface for a Syntax Model element.

C-CLEAR's generic model is based on a syntax tree data structure, and a node in the tree is defined by an `IASTNode` Java interface, show in Figure 9. As syntax trees are widely used in parsing languages, this type of data structure was chosen as C-CLEAR's common model. C-CLEAR only defines one concrete implementation of a model node which is used universally regardless of domain. Most of the operations on a node are self explanatory and include setting a parent node, adding child nodes, and setting a node value and type, defined by `INodeType` and `INodeValue` interfaces, respectively. Both the node type and node value are determined by the Domain Language Provider when the Syntax Model is constructed in the domain's Grammar Rules during the syntax parsing process. The construction of a Syntax Model by a domain will be covered in subsequent sections.

The node types allow a Domain Language Provider to define additional metadata about the node that a Domain Query Provider can later interpret. For example, a node type for CPP directives would be `CONDITIONAL_DIRECTIVE`, indicating that this node represents a conditional CPP directive like `#if`. The value of the node would typically contain actual parsed data. In the case of the `CONDITIONAL_DIRECTIVE` node type, the value would be a Boolean Syntax Tree, using the same `IASTNode` structure as the Syntax Model. To show how a Syntax Model for CPP directives would be constructed, consider the example in Figure 10, where either `WIN` or `LINUX` flags control the inclusion of `int i = 0`.

```
#if WIN || LINUX
      int i = 0;
#endif
```

Figure 10: Example used to construct a Syntax Model in Figure 11

One of the language rules defined by the CPP Language Provider is matching Boolean expressions in a directive statement like `#if WIN || LINUX`. Given a stream of tokens provided by the Lexical Tokeniser: `#`, `if`, `WIN`, `||`, and `LINUX`, the CPP Language Provider would create one model node of type `CONDITIONAL_DIRECTIVE` for `#` and `if` tokens, with a value being a syntax tree for the Boolean expression, shown in Figure 11. The controlled block would then be added as a child of the `CONDITIONAL_DIRECTIVE` node, with the value of the child being the raw conditionally compiled C code.

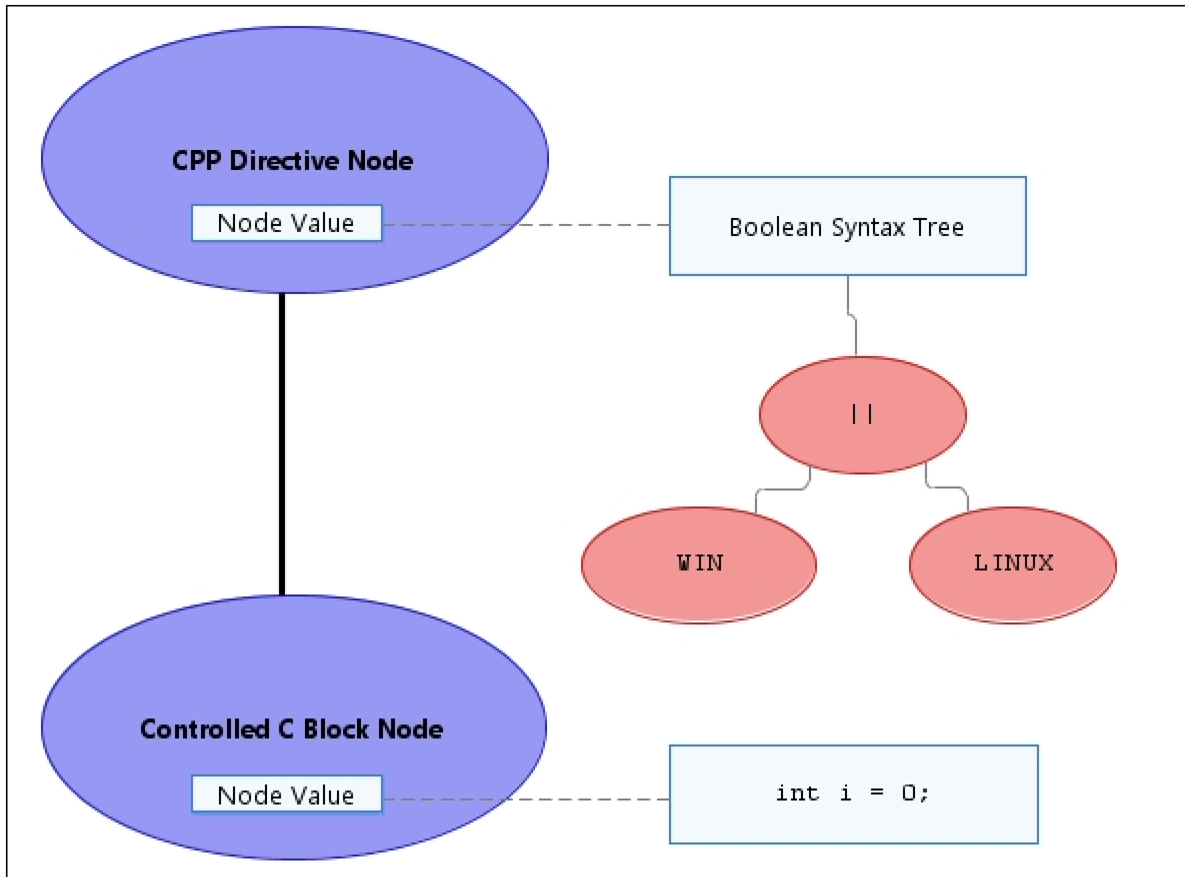

Figure 11: Syntax Model section created for the example shown in Figure 10

Further rationale to use a model based on a syntax tree is shown in the CPP schema proposed by Vidács et al. [36]. In this work, conditionally compiled code sections are child nodes of CPP conditional statement nodes. Using C-CLEAR's generic Syntax Model, a similar structure can also built as shown in Figure 11. Evaluating the degree to which other Domains can leverage the same model is part of C-CLEAR's future work.

3.1.2 Domain Language Provider

To achieve the goal of having a tool that generates a model from an input source regardless of the source domain, it is necessary to decouple the domain-specified

language Token Definitions and Grammar Rules from the parsing process, as seen in Figure 7 in Chapter 2. In terms of implementation details, C-CLEAR defines an `ILanguageProvider` Java interface, shown in Figure 12, which Domains must implement and contribute through a C-CLEAR Core extension point. The Domain Language Provider contributes Token Definitions and Grammar Rules to the framework.

Language Providers are contributed into C-CLEAR using an Eclipse extension point defined by C-CLEAR called:

`com.cclear.framework.core.language`

As an example of its usage, the contribution for parsing CPP directives is shown in Figure 14.

```
public interface ILanguageProvider extends ICCLRProvider {

      public IGrammarRules getGrammarRules();

      public ITokenDefinitions getTokenDefinitions();

}
```

Figure 12: Interface defining a language that C-CLEAR can parse.

```
public interface ICCLRProvider {

      public boolean canProvider(IInputDescriptor descriptor);

}
```

Figure 13: A light-weight API that decides if a provider should be invoked by C-CLEAR.

```
   <extension
         point="com.cclear.framework.core.language">
      <domain
            merge="false"
            priority="High"
             provider="com.cclear.domain.cpp.providers.CCPLanguageProvider"
            sourcetype="c, h">
      </domain>
   </extension>
```

Figure 14: Contribution for parsing CPP directives via an Eclipse plug-in.

Figure 15 shows the same CPP Language Provider contribution in Figure 14 using the Eclipse plug-in editor, where the Language Provider is defined by a Java class `CCPLanguageProvider` in a separate plug-in called:

`com.cclear.domain.cpp.`

As the parsing and model generation process may be long if a source project is large, a light-weight API to determine if the provider should be invoked for heavier processes is included in an interface called `ICCLRProvider` that all C-CLEAR provider interfaces extend, shown in Figure 13. Using this API, C-CLEAR can make additional decisions on whether it should start the parsing and Syntax Model generation process based on the source input type, indicated by the C-CLEAR defined Java interface `IInputDescriptor`. This input descriptor interface includes a handle to the projects containing the software system code, typically a list of Eclipse `IProject` or `IFile` resource types–the latter if only a subset of files are selected within those projects. C-CLEAR supports the analysis and data extraction of both a group of projects containing source code, or a selection of files across any of the projects. This gives a user flexibility to analyse and model only files of interest in the source code rather than the entire software system.

As multiple Domain Language Providers may be registered into C-CLEAR, the framework decides which of the providers to load based on the value of the `sourcetype` XML [58] attribute in the extension point in Figure 14. The CPP Language Provider is defined against source types “c” and “h”, corresponding to C file extensions. For example, if a source project contains a file called *thread.c*, when C-CLEAR’s Parser Controller attempts to initialise the Syntax Model Generator, it will first

check if the file extension matches any registered Language Providers by reading the `sourcetype` attribute using Eclipse platform API. If a match is found for "c" file type, the `ILanguageProvider` referenced statically in that extension XML will be loaded. In the case of the CPP domain, the language provider is `CPPLanguageProvider`, contained in the `com.cclear.domain.cpp` plug-in.

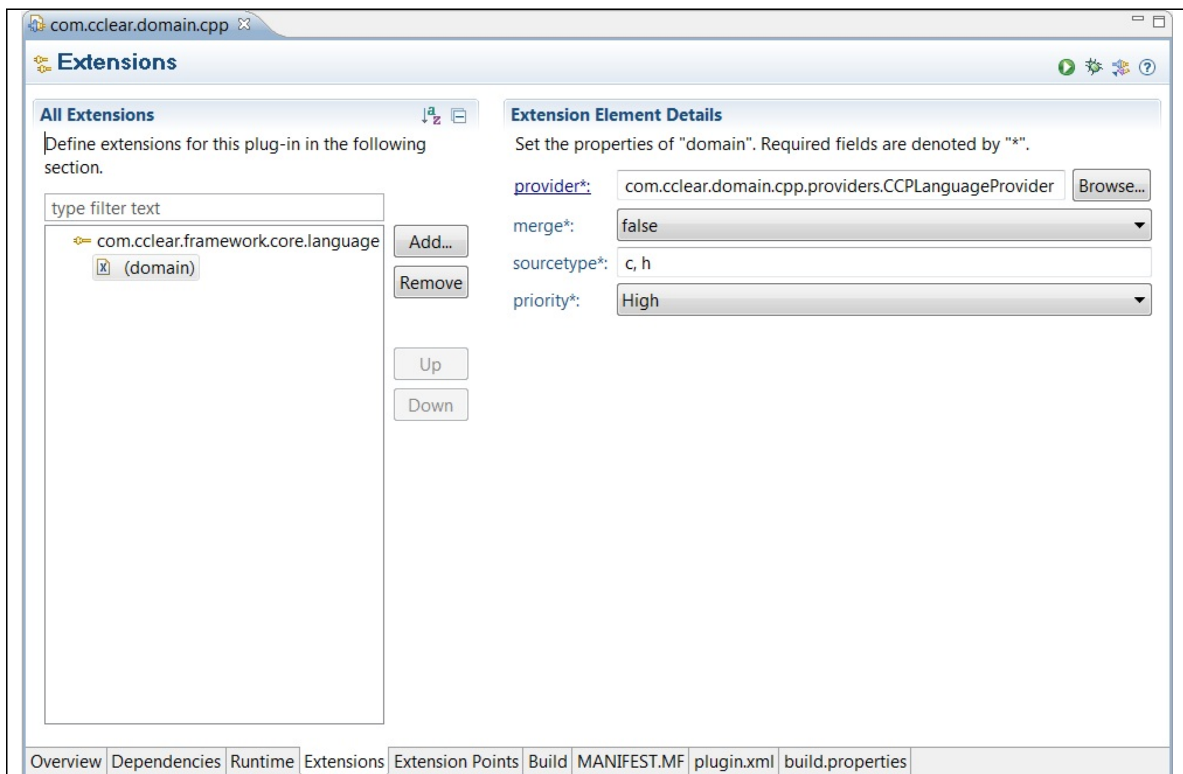

Figure 15: Contribution for CPP Language Provider via C-CLEAR Core extension point.

Once loaded, a further check via `canProvider(IInputDescriptor descriptor)` call on the provider will determine if it should be used for the actual Syntax Model generation. If the check fails, C-CLEAR will attempt to find and load another registered "c" type provider. The extension point allows multiple providers to be registered against the same file types, and a priority can be set as seen in Figure 15.

Higher priority providers are polled first, and providers of equal priority are loaded in the order that they are encountered. The loading and checking of Language Providers occurs in a Java C-CLEAR Language Provider registry called `CClearLanguageProvider`, which is used in the Parser Controller.

As C-CLEAR is only a prototype, it only supports the loading of one Domain Language Provider per parsing operation. It does not handle code bases with multiple file types that would require different Domain Language Providers, although a `merge` attribute is defined, but not fully supported where contributions from different Language Providers can potentially be merged into the same Syntax Model. The prototype implementation of C-CLEAR starts with the files in the first project in the input descriptor until one matches a registered Language Provider, and will use that Language Provider for the rest of the parsing session, skipping files that the provider cannot handle. Future versions of C-CLEAR may support multiple Language Providers in the same parsing operation as well as potentially merge results into the same Syntax Model. The merge option may also be implemented as a UI control allowing users to merge at runtime.

3.1.3 Lexical Tokeniser

This section describes the implementation of the Lexical Tokeniser, defined by an `ILexicalTokeniser` C-CLEAR Java interface, shown in Figure 16, and is responsible for generating a stream of tokens for a given file in a source project.

```
public interface ILexicalTokeniser {

    public LexicalToken next()
        throws SourceException;

    public boolean allTokensGenerated();

    public IFile getFile();
}
```

Figure 16: Lexical Tokeniser interface.

This interface is designed to be used in an iterative computation, where a new non-null token is generated every time `next()` is invoked. A null token represents the end of the token stream. The interface also provides a method to check if all tokens have been generated from the source. In addition it includes API to reference the source file that was tokenised via the `getFile()` method.

The information generated by the `ILexicalTokeniser` comes in the form of a `LexicalToken`. As there are no specialised forms of a `LexicalToken`, meaning that any token, regardless of pattern or source type, can be represented by a `LexicalToken` construct, lexical tokens in C-CLEAR are modeled by a single concrete Java type. The information carried in a token is fairly straightforward, and contains:

1. The actual value, for example: `#`, `pragma`, `ifdef`, or `LINUX32` for CPP directives.
2. Offset in the source where the token is found. The token length is not explicitly contained as it can be derived from the length of the value. The offset in source, on the other hand, is important for IDE integration, as it allows navigation to that particular source location in an editor, should the token value appear as a result in the Query Results View.
3. Type of token, which is defined by a Language Provider's Token Definitions.

For CPP directives, the CPP Language Provider defines token type categories which C-CLEAR can understand. The categories include keywords and operators used in Boolean expressions in conditional statements such as `#if`. Within the CPP Language Provider, these token types are defined using integers, which then get used as token types in a `LexicalToken`.

These integer values get mapped to the actual patterns shown in Appendix Table 7, and this map is part of the token contribution that a Language Provider must define, as it is used during the tokenising process in the Lexical Tokeniser.

3.1.4 Syntax Parser

The Syntax Parser is responsible for generating a Syntax Model based on the stream of tokens provided by the Lexical Tokeniser and the Grammar Rules of a Language Provider, and is defined by the Java interface, `ICCLRParser`, shown in Figure 17**.** The Syntax Model in C-CLEAR is contained in a `TranslationUnit` which has a reference to the root node of the model. In addition it includes any errors or warning markers that were generated while creating the model.

```
public interface ICCLRParser {

	public TranslationUnit resolve() throws SourceException;

	public List<ParserMessage> getErrors();

	public boolean hasErrors();

}
```

Figure 17: Interface defining a Syntax Parser in C-CLEAR.

A `TranslationUnit` is the principle output of C-CLEAR's Syntax Model Generator component, and contains the Syntax Model for a parsed source. The API

hides the fact that a Lexical Tokeniser is required in the parsing process. The only information required from the parser is a generated model, along with possible errors that occurred during the parsing process. In certain cases, critical failures may be encountered that result in no model, as opposed to a model with errors, in which case the API throws a checked exception. A possible case when an exception is thrown is if no Language Provider is found for a given source type, or an I/O error occurred while reading the source files and a partial model cannot be created.

Otherwise, when a partial model is constructed with errors, possibly due to certain grammar rules having been violated, the API allows the retrieval of all errors in the form of a list of parser messages. Parser messages contain information about the error, including the line number in source file where the error was encountered as well as a serialised message of the error generated by some domain grammar rule that was not matched.

During the parsing process, C-CLEAR will utilise the same Language Provider used in the Lexical Tokeniser and ask for its Grammar Rules, which is implemented by the domain using the `IGrammarRules` interface, shown in Figure 18.

```
public interface IGrammarRules {

	public IASTNode generateGrammarResult(
			ITokenBuffer tokens, IASTNode parent);

	public boolean markIndexConstruct(LexicalToken token);


	public List<ParserMessage> getErrors();

}
```

Figure 18: Interface that a Language Provider must implement in order to construct Syntax Model nodes and mark constructs for the C-CLEAR index.

The Syntax Parser is fed a stream of tokens from the tokeniser through a lookahead mechanism [57, 37], defined by an `ITokenBuffer` interface, where *n* number of tokens are buffered sequentially. The lookahead buffer is then passed on to a Language Provider's Grammar Rules to generate a Syntax Model via

```
generateGrammarResult(
                ITokenBuffer tokens, IASTNode parent)
```

as shown in Figure 18.

This means that the construction of the Syntax Model is delegated to the Language Provider, allowing the domain to establish the structure of the model based on Grammar Rules that define the language. The Syntax Model in C-CLEAR is simply another abstraction of a given source, and in some cases provides a light-weight alternative to the heavier structures generated by an IDE language tooling component like CDT, in particular for tools only intending to analyse portions of a source code.

Figure 19 shows one of the simplified grammar rules for CPP domain implemented in the CPP Language Provider's Grammar Rules, which is responsible for matching conditional CPP directives in a C file. The rule is defined by the Extended Backus–Naur Form [22]:

cSection ::= block cSection | condCompSection cSection | ε

A cSection is a section of C code defined by either a block of common C code (non-conditionally compiled) or a conditionally compiled section of code followed by further cSections.

The lookahead buffer allows tokens to be cached *n* levels deep to allow grammar rules to check token *m*, where:

$n > 0 \wedge m > 0 \wedge m <= n$

For example, consider `#if WIN`. The tokeniser will generate three tokens, `#`, `if`, and `WIN`. However, in order for the language rule that defines `#if` to be matched, the lookahead buffer has to be at least 2 levels deep to contain the `#` and `if` tokens. Containing just the `#` is too ambiguous as there are multiple language rules that start with the pound sign.

The lookahead buffer allows domain Grammar Rules to check for an ordered sequence of tokens before finding the grammar rule that satisfies that sequence, and prevents the unnecessary and incorrect consumption of tokens from the lookahead buffer in case a grammar rule is mismatched. As a grammar rule consumes tokens from the lookahead, the lookahead buffer automatically fills itself via call backs to the Lexical Tokeniser until all tokens have been parsed from the input stream.

Figure 19 shows the grammar rule looking ahead by 1 element to identify a `#` token. If it is identified, the token is not yet consumed as no language rule has been satisfied yet. Rather, the rule will look ahead to the next token via:

```
if ((nextToken = lookNext(2)) != null) {…}
```

to check to see if the next token is something it recognises like `if`, `ifdef`, `ifndef`. If so, it proceeds to invoke another grammar rule that handles the actual conditional CPP statement via a call to `cppCondCompSection(parent)`.

The rule also handles error conditions, in particular if a # token is encountered, but the next token is not recognised as a CPP directive keyword via a `handleError(..)` call which simply stores errors for later retrieval by the `getErrors()` API shown in Figure 18.

Note that the CPP Grammar Rules relies on token type definitions which were previously contributed by the same CPP Language Provider through its `ITokenDefinitions` implementation, which is evident in the `nextToken.getType()` call in Figure 19.

This section has shown three key components in the Syntax Model Generation component working together via an intermediate lookahead token buffer. The Lexical Tokeniser is responsible for generating `LexicalTokens` based on `int` value types obtained from the CPP Language Provider during the tokenising process. C-CLEAR's Syntax Parser is then responsible for creating a lookahead buffer that contains a reference to the Lexical Tokeniser, and which then gets passed into a Domain Language Provider's Grammar Rule. It is up to the Language Provider to construct the Syntax Model based on the token types it earlier had defined in the `ITokenDefinitions` implementation.

C-CLEAR has no knowledge of domain token types. The framework is responsible for generating and buffering tokens into a lookahead buffer, while the domain defines the token types and recognises them in its Language Provider Grammar Rules, as seen in Figure 19. This shows how domain specific logic and semantics are contained on the domain side, while actual functionality such as generating and buffering a token is handled by the framework side. The domain is then responsible for consuming tokens from the buffer and matching them against a grammar rule, and decides the structure of the Syntax Model. C-CLEAR is in charge of any management of the Syntax Model once the model has been created by the domain Grammar Rules.

```
private void cSection(IASTNode parent, int nesting) throws SourceException {

        if (reachedErrorThreshold || (parent == null)) {
                return;
        }

        /*
         * Each iteration of this method creates a temporary subtree, whose
         * children get merged with the subtree created in the stack frame below
         * the current one (i.e. the caller of this method), assuming this
         * method is being called recursively.
         */
        if (lookAheadBuffer.moreToProcess()) {
                LexicalToken nextToken = lookNext(1);
                /*
                 * If the nextToken in the lookahead is still null, but the end of
                 * the input source has not been reached, an empty line has been
                 * encountered.
                 */
                int nextTokenType = (nextToken == null) ? EMPTY_LINE_SYM
                                    : nextToken.getType();

                if (nextTokenType == POUND_SYM) {
                        int lineNumberError = nextToken.getLineNumber();
                        if ((nextToken = lookNext(2)) != null) {
                                nextTokenType = nextToken.getType();
                                if (nextTokenType == IF_SYM
                                            || nextTokenType == IFDEF_SYM
                                            || nextTokenType == IFNDEF_SYM) {
                                        cppCondCompSection(parent);
                                        cSection(parent, nesting);
                                } else if (nesting == TOPLEVEL_EXPRESSION) {
                                        handleError("Wrong directive at line "
                                                        + lineNumberError,
                                                        lineNumberError);
                                }
                        } else {
                                handleError("Missing directive at line " + lineNumberError,
                                                lineNumberError);
                        }
                } else if (nextTokenType == EMPTY_LINE_SYM) {
                        parent.insert(createNode(AST_NODE, BLOCK, blockOfLines()));
                        cSection(parent, nesting);
                } else if (nesting == TOPLEVEL_EXPRESSION) {
                        int lineNumber = nextToken.getLineNumber();
                        handleError("Wrong token at line " + lineNumber, lineNumber);
                }
        }
}
```

Figure 19: CPP directives grammar rule *cSection ::= block cSection | condCompSection cSection* / ε

3.1.5 Tokenising and Parsing Summary

This implementation shows a separation between data extraction and buffering which falls under C-CLEAR's control, and the definition of tokens and Syntax Model structure, which is the responsibility of the domain. C-CLEAR does not inherently know how to construct the Syntax Model, so it delegates that operation to the Domain

Language Provider. However, the provider does not need to handle token generation and buffering from an input. It simply needs to define a set of token types, allow C-CLEAR's Lexical Tokeniser and Syntax Parser to create and populate a lookahead buffer, and rely on consumption of tokens from the lookahead buffer to match against grammar rules. This allows the tool developer to focus solely on domain implementation rather than the parsing mechanism from an input.

3.1.6 Construct Marking and Indexing

Abstraction and conceptualisation of data was identified as an important feature of a software maintenance and analysis tool in Section 1.2.3. The prototype version of C-CLEAR defines a common data abstraction mechanism, where the framework allows a domain to mark certain facts of parsed data as constructs that can then be displayed in the Constructs View. The constructs represent some abstraction that can aid a user in deciding what portions of a software system to query and analyse. The prototype version of C-CLEAR does not support dependencies or relationships between these constructs, but it will be an important future enhancement, as abstractions may contain relationships of interest to a user and allow for further extraction of higher level concepts.

For the CPP domain, these constructs are marked by the CPP Language Provider's Grammar Rules via implementation of the `markIndexConstruct(LexicalToken token)` API, shown in Figure 18, during the syntax parsing process. Since the Domain Language Provider is responsible for defining lexical tokens, it may easily determine if the token type processed during the framework parsing stage can be conceptualised. In the case of the CPP Language

Provider, any token of type IDENTIFIER is marked as a construct, as they represent CPP flags controlling conditionally compilation.

Therefore the implementation of markIndexConstruct(LexicalToken token) for the CPP Grammar Rules is a simply a one line check in the return statement on the LexicalToken type against IDENTIFIER. If it matches, true is returned, or false otherwise, as seen in Figure 20. The token value, which would happen to be CPP flags like LINUX32 is then added into C-CLEAR's index as a key along with the Syntax Model node in which the construct was contained as the value. As constructs may occur in multiple model nodes, the index entry value is in fact a set of Syntax Model nodes.

```
public boolean markIndexConstruct(LexicalToken token) {

      return token != null
              && token.getType() == IDENTIFIER;


}
```

Figure 20: Simple implementation for CPP construct marking.

In the case of the CPP domain, flags are marked as constructs as they may represent types of configurations for a C source. C-CLEAR's generic data abstraction mechanism is a first pass prototype that will undergo significant changes in future versions. For now, the mechanism is tied to a simple API in the Language Provider's Grammar Rules. It will be extended to eventually support construct dependencies and relationships.

3.1.7 Parser Controller

The main controller in C-CLEAR Core that initiates the parsing process is the Parser Controller, shown in Figure 7 in Chapter 2, and implemented by `CCLRParser` Java class. Figure 21 shows the initialisation method that instantiates several components used during the parsing process A new Parser Controller is created every time a parsing request is made from the UI, and includes creating the construct index and constructs controller, `ConstructIndex` and `Constructs`, respectively, as well as configuring the Lexical Tokeniser via a `LexicalTokeniserConfig`. It also includes an optional lexical filter which contributors to the framework can implement that will omit tokenisation of certain data in a given domain. The purpose of this filter is to allow further specialisation within the same domain without having to implement a completely separate Language Provider for just minor variations. For example, in the CPP domain, a more specialised version of this provider would just want to focus on parsing `#pragma` and omit `#if`. Instead of contributing a separate CPP Language Provider, a domain would simply have to contribute an additional filter using the same CPP Language Provider. However, this feature was not extensively used in C-CLEAR therefore will not be covered in greater detail.

The Parser Controller shows an important aspect of the C-CLEAR framework, which are domain contributions. When C-CLEAR receives a request to parse an input, the first step it takes is to attempt to load a Language Provider and obtain token definitions via a `loadLanguageTokenTypes(IInputDescriptor)` call. Loading a Language Provider involves a request to the C-CLEAR language registry, `CClearLanguageRegistry`, to find registered Language Providers based on the

input type. If the initialisation process fails to find a token definition, initialisation stops and an error logged. Otherwise, C-CLEAR proceeds to load a lexical filter, which may be null, although no null check is necessary as it is an optional configuration. It then continues on to create a lexical configuration based on the language token types loaded from a Language Provider. The lexical configuration contains all the necessary information and components to start the parsing process, and is used only when a parsing request is actually made.

There are four main components to the indexing mechanism in C-CLEAR. The `CCLRASTIndexer` is a specialised index that indexes the Syntax Model per file. This allows rapid access to corresponding portions of the Syntax Model during file level operations, for instance synchronising the model when changes occur in an IDE file editor. Note that this is separate from the construct index, which indexes abstract constructs marked by the Language Provider during the syntax parsing process.

The second component of the indexing mechanism is the `ConstructIndex` and is passed to a third component, the construct controller, called `Constructs`, which manages event handling and communication between the index and the parser. The fourth component is a task that contains additional API and allows the parser to communicate with the `ConstructIndex` while Syntax Model is being generated.

```
protected boolean initialiseParser(IInputDescriptor inputDescriptor) {

        // First find a token definition from the language registry
        ITokenTypes languageTokenTypes = loadLanguageTokenTypes(inputDescriptor);

        if (languageTokenTypes == null) {
                CCLRCorePlugin.getInstance().logError(
                                ERROR_NO_LANGUAGE_PROVIDER_FOUND);
                return false;
        }

         boolean generateTokenForEmptyLine = shouldTokeniseEmptyLine();

         IlexerFilter lexerFilter = getLexicalFilter();
         lexicalTokeniserConfig = new LexicalTokeniserConfig(
                                languageTokenTypes, lexerFilter,
                                generateTokenForEmptyLine);
        astIndexer = new CCLRASTIndexer();
        ConstructIndex indexer = new ConstructIndex();
        cclrIndexer = new Constructs(indexer);
        task = new SyntacticIndexingTask(indexer);

        syntaxParserConfig = getLanguageProviderParserConfiguration();
        return true;

}
```

Figure 21: Initialisation of Syntax Model Generator components in the Parser Controller.

Parser Controller initialisation is lazily performed only when C-CLEAR’s Core layer receives a request to generate a Syntax Model from the UI via the Parser Controller’s `generateAST(IInputDescriptor input)`method, as shown in Figure 22. It will first attempt to initialise the Parser Controller, and if it fails due to not finding a token definition for the given input from a Domain Language Provider, the parsing process halts. Otherwise, the Syntax Model generation process continues with creating the parsing components and finally generating the Syntax Model via a call to `parser.resolve()`.

```
public synchronized TranslationUnit generateAST(IInputDescriptor input)
               throws SourceException {
        if (!initialiseParser(input)) {
               return null;
        }

        TranslationUnit transUnit = null;
        Reader reader = SourceInputFactory.getReader(input);
        if (reader != null) {
               IlexicalTokeniser tokenStream = getLexicalTokeniser(input, reader);
               ICCLRParser parser = getSyntaxParser(input, tokenStream);
               transUnit = parser.resolve();

               astIndexer.getIndex().put(input, transUnit);
        }
        return transUnit;
}
```

Figure 22: Executing the parsing request via the Parser Controller. Note that if initialisation fails, parsing is terminated.

3.2 User Interface

C-CLEAR's UI is general and meant to be reused regardless of the domain being analysed. Table 2 in Chapter 2 lists four main UI components of C-CLEAR which are implemented in the plug-in `com.cclear.framework.ui.` Screenshots of all four UI components are shown below, starting with the Project Selection feature, shown in the UI portion of the architecture diagram in Figure 7.

3.2.1 Project Selection

Figure 7 in Chapter 2 shows the Project Selection UI component, which forms part of C-CLEAR's integration into Eclipse. The Project Selection is implemented as an Eclipse Navigator context menu action, which initiates the parsing processing on a project selection in the Eclipse Project Explorer view. Figure 23 shows the context menu action invoked on a Kaffe JVM project.

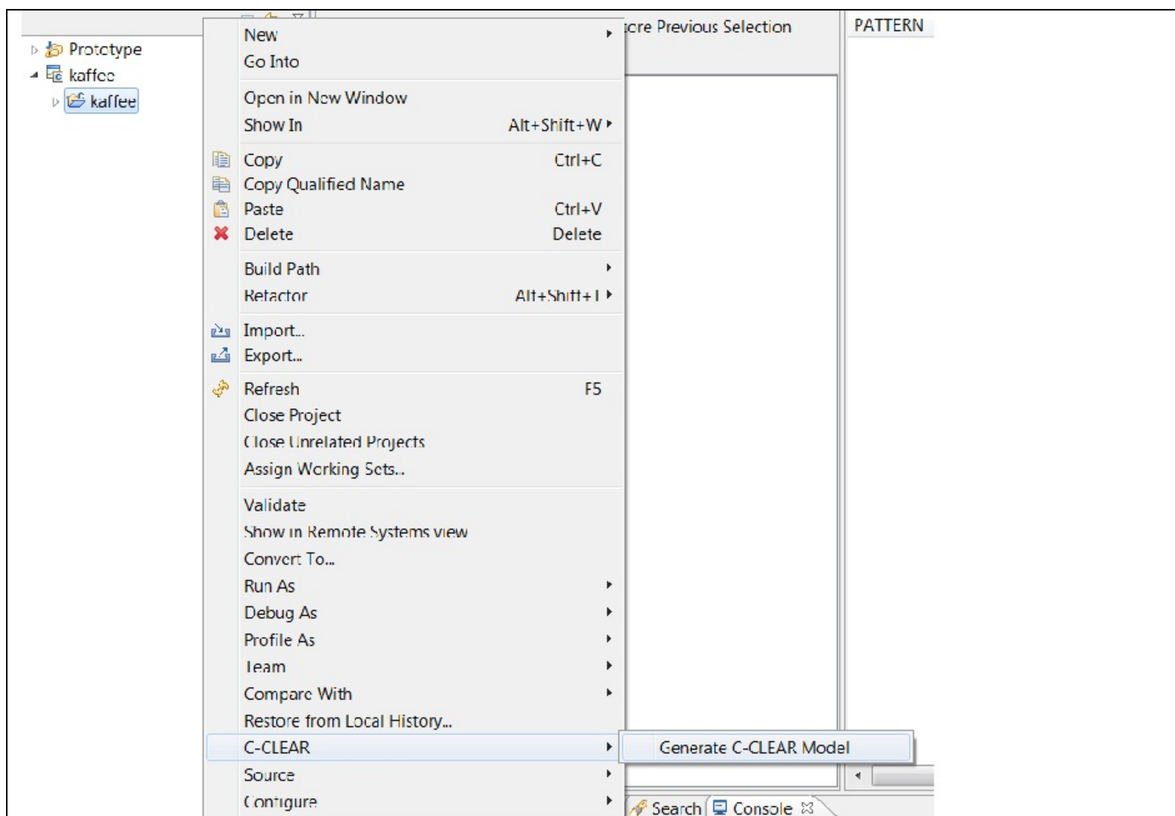

Figure 23: C-CLEAR context menu action that invokes C-CLEAR on a project selection, in this case, Kaffe JVM.

3.2.2 Constructs View

The Constructs View is one of two UI views defined by and implemented for C-CLEAR that integrates into the Eclipse workbench, and displays indexed constructs as marked by a Domain Language Provider during the parsing process. It also has a general look and feel expected from an Eclipse view. The intention is to show marked concepts in the parsed data that allow for better understanding of a particular concern in a software system, and permit the user to more concisely query and analyse a code base. Section 1.2.3 in Chapter 1 indicated that some level of abstraction is important for a software analysis tool, particularly one that analyses source code. However, for the prototype version of C-CLEAR, the initial implementation of the Constructs View only contains a

selectable table, where constructs are displayed in an alphabetically ordered list, and any semantic significance is carried in the construct's name. No relationships between these constructs are shown, or any possible conceptual grouping, but this could be implemented in future versions of C-CLEAR. In addition, the Constructs View could be complemented with a Software Visualisation view that will present the same constructs, along with any possible grouping or dependencies, in a graph type yet to be determined. Figure 24 shows the Constructs View displaying CPP flags that control conditional compilation for the Kaffe JVM.

Once a C-CLEAR parsing operation has been initiated on a set of projects in Eclipse, the domain-marked constructs are displayed in the Constructs View. In the case of the CPP domain, these constructs are the CPP flags that are marked by the CPP Language Provider during the Syntax Model generation process. The flags are only shown in the view after the Syntax Model has been generated.

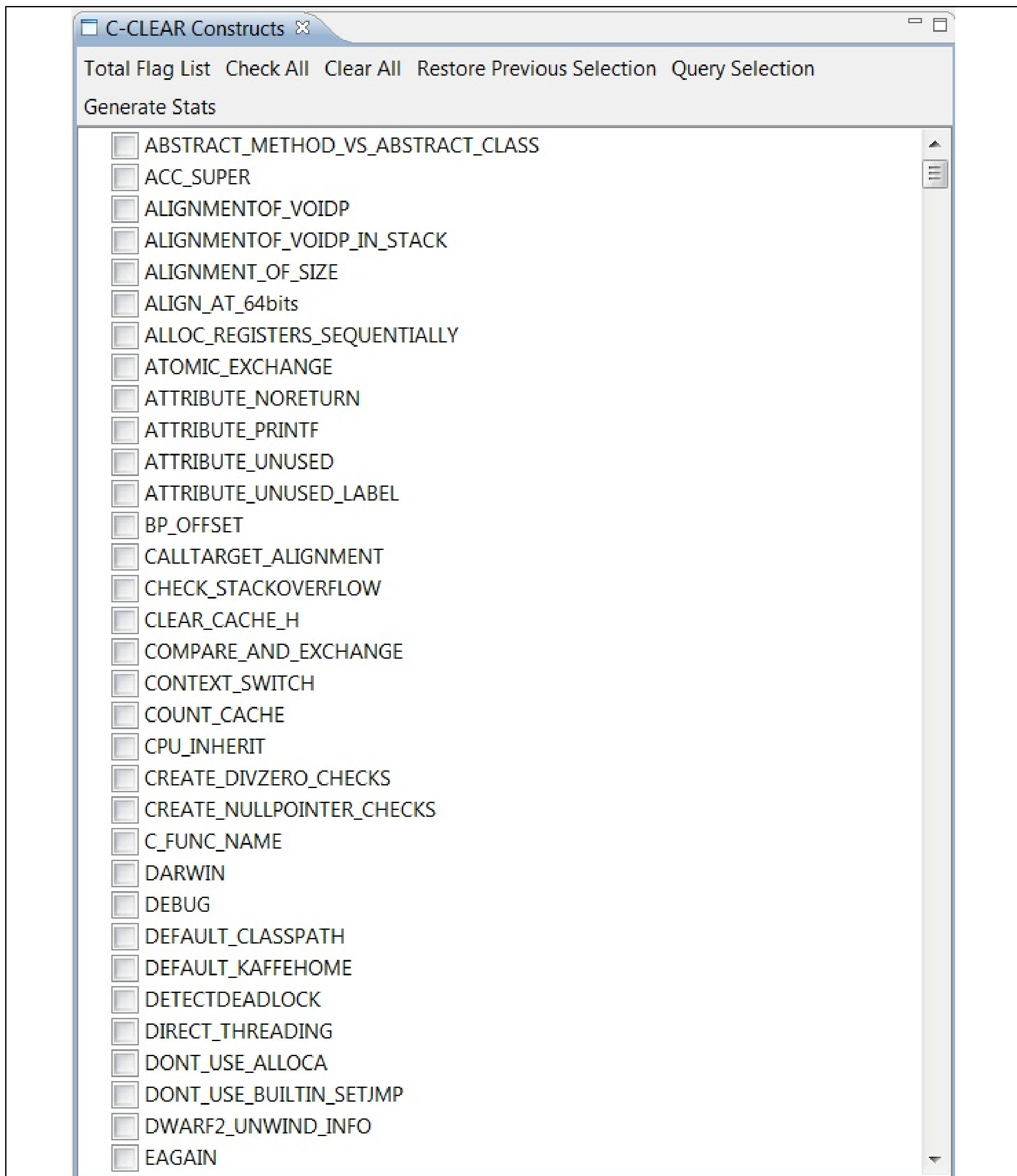

Figure 24: C-CLEAR Constructs View showing indexed constructs for Kaffe JVM, which in this case are CPP directive flags.

The Constructs View allows users to select constructs to query. It also keeps an immediate history of the last selection of constructs selected for the query. Ideally the history should maintain *n* number of selections, preferably accessible through a drop-

down menu, where *n* is a user assigned parameter that can be set in Eclipse's preferences pages. However, as the tool is a prototype, only one immediate history is recorded, and accessible via a Restore Previous Selection button.

Query operations are also launched from the Constructs View through the Query Selection button, and results are displayed in a separate Query Results View, as seen in Figure 25. Ideally, a Domain Query Provider should be able to contribute a set of queries, and the user should be able to select a query to perform from the Constructs View. However, as C-CLEAR is only a first pass look a tooling frameworks, the framework was designed to accept only one query per domain. Future work on C-CLEAR will allow Domain Query Providers to contribute a set of queries that can be selected through a drop-down menu in the Constructs View.

3.2.3 Query Results View

The second view implemented by C-CLEAR is the Query Results View, which displays the results of a query launched from the Constructs View, shown in Figure 25. As with the Constructs View, it maintains a common IDE look and feel. In this case, the query contribution from the CPP Query Provider will find all conditionally compiled blocks of code controlled by the selected flags in the Constructs View. Once C-CLEAR obtains all the results from the CPP query, it will display them in the Query Results View. The displayed data includes general, domain independent properties like number of occurrences for each result and location information such as file names and line numbers, and the actual result value, which in the case of the CPP domain is raw conditionally compiled C code, as seen in Figure 25. C-CLEAR assumes that all parsed data in a Syntax Model are facts obtained from a file, therefore every model node value

contains general properties like file and line number location. This data is obtained for any token modeled into a Syntax Model element during the parsing process, and is therefore domain independent. This allows C-CLEAR to implement common navigation to IDE components like source file editors regardless of the parsed domain, and with no semantic knowledge of what the Syntax Model elements represent. The ability to keep track of general properties of parsed data regardless of the software system that is being analysed is a critical feature of C-CLEAR that allows integration into the IDE.

The Query Results View shows the affected line number containing the result, and double clicking on any entry in the view will navigate to that location in the file. C-CLEAR does not define a source file editor. Rather, it will open an appropriate Eclipse editor based on the file type containing the result. In the case of the CPP query, selecting a query results will open the Eclipse C editor and highlight the location of the query results in the editor.

The Query Results View is implemented using a JFace [31] table tree viewer, where the root node in the first column of each row shows the actual result. In the case of the CPP query, the result is the raw conditionally compiled section of code controlled by the selected flags in the Constructs View. Figure 25 shows a query executed on `WIN32` and `_GNU_SOURCE` flags. The query for the CPP domain was implemented using a Levenshtein algorithm [6] to seek conditionally compiled code blocks that are syntactically equivalent. Four results are found, two for each conditionally compiled pattern. Expanding each root node shows the files in which the result was found, and further expansion shows the line numbers for each occurrence of that result. Double clicking on any of the line number entries will navigate to the specified location in a C

editor. The view also displays the total number of matches per result as well as the total number of files containing the results in two separate columns. All three columns can be sorted in ascending or descending order.

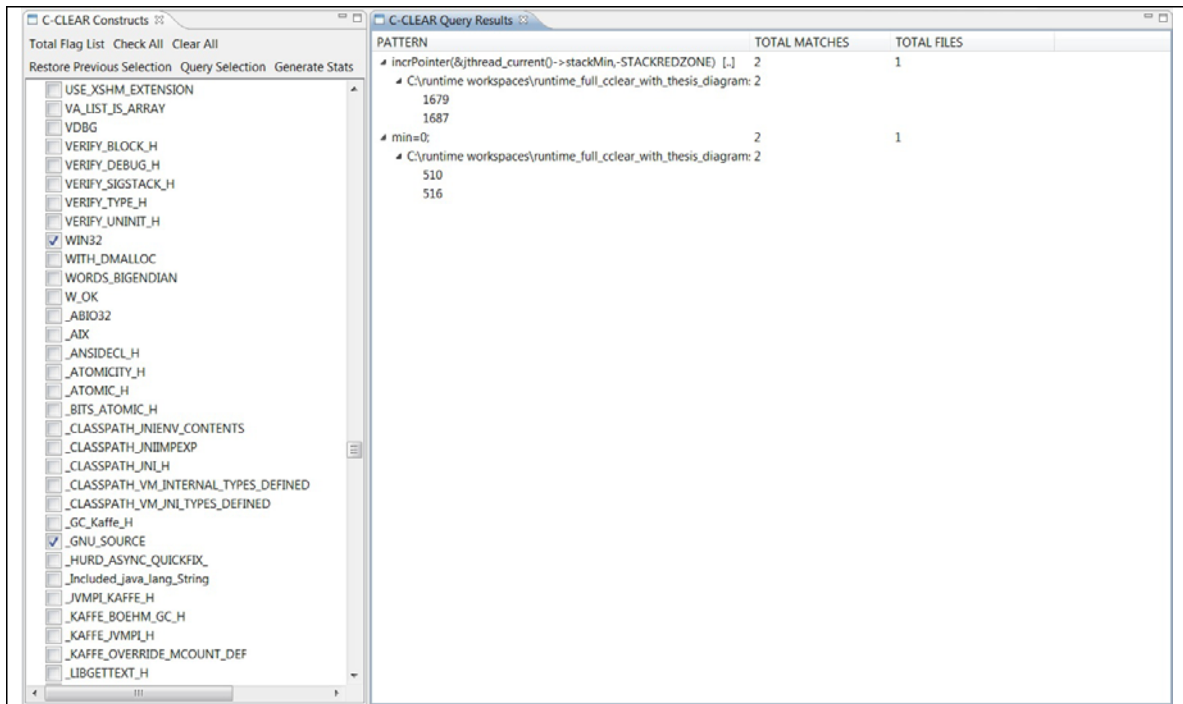

Figure 25: C-CLEAR Query Results View showing the results of a CPP query for finding duplicate code blocks controlled by the selected CPP flags.

3.2.4 Query Preferences Page

The range, scope and accuracy of a query can be configured through a C-CLEAR Query Preferences Page, which integrates into Eclipse's Preferences Page, shown in Figure 26. This query configuration feature of C-CLEAR follows similar principles as mentioned in Section 1.2.2 in Chapter 1, where tools like Motive implement a slider that can adjust scope and range of a query. As with all other components of C-CLEAR, these values are meant to be generic and interpreted by the domain query during a query

execution in a manner that makes sense to the domain. The query parameters can also be ignored by the domain query, in which case they will have no effect. In addition, query results can be serialised into a file, and the preferences page allows users to select a location for the file. The other three query parameters, *Deviation*, *Minimum number of results within deviation*, and *Maximum number of results within deviation* will be covered in greater detail in Section 3.3.

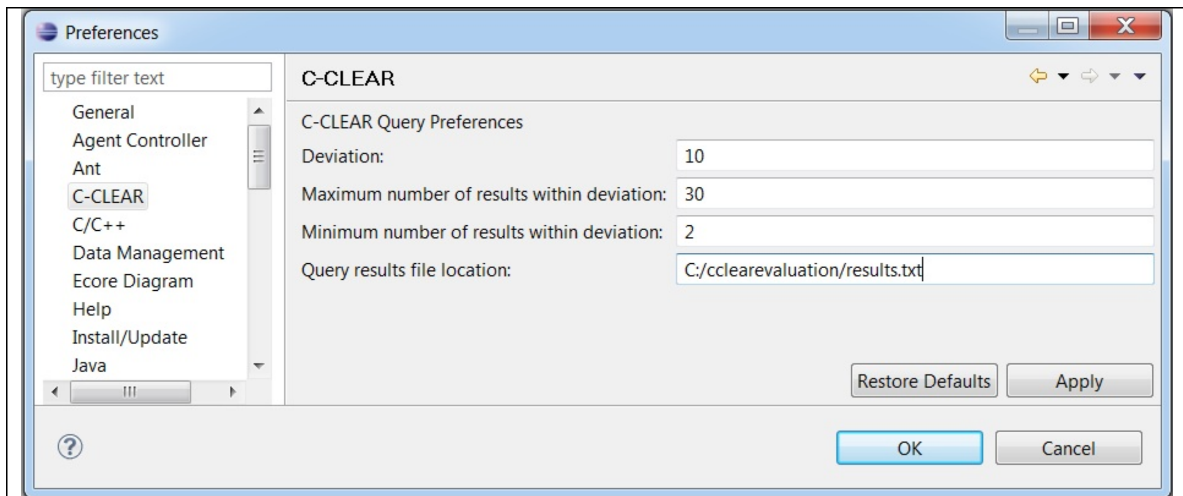

Figure 26: C-CLEAR query preference page that allows query parameters to be configured.

3.3 Query

Sections 3.2.3 and 3.2.4 showed the UI for displaying query results and configuring queries. Query configurations are meant to be generic and left to the domain query to interpret and adjust query results accordingly. This section will describe the Query portion of the C-CLEAR framework in greater detail.

3.3.1 Domain Query Provider

The Query framework component of C-CLEAR shown in Figure 7 in Chapter 2 is implemented in a plug-in called `com.cclear.framework.query`. In this section, the primary function of the common Query mechanism is described, where a query from a domain is loaded, executed, and results are added to an internal `ResultsTable` structure by a Query Manager in the C-CLEAR Query framework component.

Queries are executed in the framework by walking the Syntax Model, performing the query operation on each model element that contains the constructs selected for the query. Recall that C-CLEAR indexing mechanism maintains an index of constructs and their containing Syntax Model elements, therefore queries may be performed relatively quickly by simply looking up the list nodes per selected construct, and passing that list to the query provider for the language. The domain query then visits those nodes and performs actions specific to that domain, for instance further evaluating CPP directives in nodes that map to the selected CPP flags in the Constructs View in the case of the CPP domain.

In terms of implementation, a query request from the UI is handled by the Query Controller, and invokes `runQuery(..)` on the Query Manager, seen in Figure 28. Figure 28 also shows the actual loading of a query via `getQuery(..)` call where C-CLEAR loads a query from the same domain that was used in the Syntax Model generation process. A domain defines a query by implementing the `ICCLRQuery` interface shown in Figure 27 and contributes it into C-CLEAR through a Domain Query Provider.

```
public interface ICCLRQuery {

      public void evaluate(INodeValue value,
                           IQueryDescriptor descriptor);

}
```

Figure 27: Interface that must be implemented by a domain and contributed by a Query Provider.

The query evaluates a Syntax Model node value and adds the results to a query descriptor. The API shown in Figure 27 is an early version of the query interface. Ideally, the evaluation API should be returning the result as opposed to relying on a query descriptor to pass on the results to C-CLEAR. Future work on C-CLEAR will address this API shortcoming.

In Figure 28, once a query has been loaded, it is executed in an `executeQuery(..)` method in the Query Manager, shown in Figure 29 which takes in the constructs that were selected in the Constructs View.

```
public void runQuery(int queryMargin, ResultsTable table,
                      List<IInputDescriptor> input,
                      List<String> constructs,
                      CCLRParser parser) {

  if (!initialiseTable(table, input, constructs, parser)) {
    return;
  }

  try {

      ProgressMonitorDialog progress = new ProgressMonitorDialog(
                        PlatformUI.getWorkbench()
                                  .getActiveWorkbenchWindow()
                                  .getShell());
       IASTWalkerTask evaluatorTask = getASTTask(constructs);

       ICCLRQuery query = getQuery(table, queryMargin, maxBlockSize);

       executeQuery(query, evaluatorTask, progress, input, parser);

   } catch (InvocationTargetException e) {
     QueryPlugin.getDefault().logError(e);
   } catch (InterruptedException e) {
     QueryPlugin.getDefault().logError(e);
   }
}
```

Figure 28: A query request is processed by the Query Manager via the `runQuery(..)` API.

During the execution of a query, a `QueryTask` is created and passed on to an `ASTWalker`, whose job is to walk the Syntax Model and execute the query via a call to `traverseAST(…)`.

Figure 30 shows an `evaluateCPPNode(..)` implementation for the CPP query that calculates whether conditionally compiled sections of code are syntactically similar. Internally the query maintains a table of results, which it uses when comparing new sections of conditionally compiled code.

As the query is invoked while the Query Manager walks the Syntax Model, if the CPP query encounters a node value of type `CPPSymbols.BLOCK,` it will obtain its value, defined by a `LineBlock.` A line block is defined by the CPP domain, and is simply a raw, unparsed section of C code that is conditionally compiled, but does not include the CPP directives. Figure 11 in Section 3.1.1 shows a child node added to the CCP directives node. The value of the child node is a `LineBlock` with a value `int i = 0;`.

The CPP query uses this CPP controlled code block for syntactic comparison using an `ISemanticAnalyser` interface. The semantic analyser is simply a comparator with a `compare(..)` method which compares two String values for similarity.

The CPP query uses a Levenshtein implementation of the analyser, which it loads via a `getAnalyser(..)` call, as seen in Figure 30. The actual Levenshtein algorithm implementation is shown in Appendix Figure 43, where the distance or deviation between characters in both Strings is computed. The larger the deviation, the lower the syntactic equivalence between the two Strings.

```
protected void executeQuery(ICCLRQuery query, final IASTWalkerTask task,
                      ProgressMonitorDialog progress,
                      final List<IInputDescriptor> finInput,
                       final CCLRParser finParser)
                              throws InvocationTargetException,
                                     InterruptedException {
  progress.run(true, true, new IRunnableWithProgress() {

      public void run(IProgressMonitor monitor)
                                    throws InvocationTargetException,
                                           InterruptedException {
        try {
                int monitorWork = finInput.size();
                monitor.beginTask("Parsing and Querying", monitorWork);
                for (Iterator<IInputDescriptor> it = finInput.iterator();
                   it.hasNext() && !monitor.isCanceled();) {
                        IInputDescriptor inputDesc = it.next();
                        if (inputDesc != null) {
                                TranslationUnit translationUnit =  (TranslationUnit)
                                    finParser.getCurrentAST(inputDesc);
                                if (translationUnit != null) {
                                        ASTWalker walker = new ASTWalker(
                                                              new QueryTask(
                                                               inputDesc,
                                                               query,
                                                               task));
                                        traverseAST(walker, translationUnit);
                                }
                        }
                        monitor.worked(1);
                }

                monitor.done();


        } catch (SourceException e) {
                QueryPlugin.getDefault().logError(e);
        } catch (MinerException e) {
                QueryPlugin.getDefault().logError(e);
        }
    }
  });
}
```

Figure 29: Actual query execution using a Syntax Model visitor task that evaluates nodes of interest to the query.

Recall that in Section 3.2, query comparisons can be configured via the UI in the Query Configuration Preferences page, shown in Figure 26. A *Deviation* parameter in the preferences page allows Query Providers more flexibility to adjust the range or scale of the results based on user input, similar to what other tools such as Motive support. However, the field is not actually used by C-CLEAR framework, but rather left for interpretation to the domain query when performing a query operation. This generic mechanism allows a domain to react to user preferences. If not used by a domain query,

the values are ignored. In the case of CPP domain, the CPP query uses the *Deviation* parameter as an upper bound limit when computing a Levenshtein distance between two C line blocks. If the distance is less than or equal to the *Deviation* then the two String patterns are considered to be syntactically similar and added to the results table as seen towards the end of the `evaluateCPPNode(..)` method in Figure 30.

```
protected void evaluateCPPNode(INodeValue value, IQueryDescriptor descriptor) {

        if (value == null || descriptor == null
                        || value.getType() != CPPSymbols.BLOCK) {
                return;
        }

        int maxBlockSize = descriptor.getMaxBlockSize();
        int distance = descriptor.getDistance();

        ISemanticAnalyser analyser = getAnalyser(maxBlockSize);

        LineBlock block = (LineBlock) value.getValue();
        String fileName = descriptor.getFileName();

        if (block == null || fileName == null) {
                return;
        }
        StringBuffer bufferToCompare = completeBuffering();
        if (bufferToCompare != null && (bufferToCompare.length() > 0)
                                && (bufferToCompare.length() < maxBlockSize)) {
                        boolean added = false;
                        CCLRFile file = null;
                        String stringToCompare = bufferToCompare.toString();

                        for (Iterator<?> it = table.iterator(); it.hasNext() && !added;) {
                                MinedPattern pattern = (MinedPattern) it.next();

                                String patternString = pattern.getPattern();
                                if (Math.abs(patternString.length()
                                                - stringToCompare.length()) <= distance) {
                                        if (analyser.compare(patternString, stringToCompare)
                                                                <= distance) {
                                                added = addMatchedPatternFile(fileName,
                                                                block, added, pattern);
                                        }
                                }
                        }

                        if (!added) {
                                file = new CCLRFile(fileName);
                                file.add(block);
                                MinedPattern pattern = new MinedPattern(stringToCompare);
                                pattern.add(file);
                                table.add(pattern);
                        }
        }
        clear();
}
```
Figure 30: CPP query for comparing conditionally compiled code blocks.

A minimum and maximum number of query matches within a deviation threshold can also be specified in the preferences page. As with the *Deviation* parameter, both are ignored if not used by a domain query. A domain can use these as lower and upper bounds when deciding whether to add query results to the `ResultsTable`. Although not shown in Figure 30, as an example, if the minimum value is set to 2, the CPP query will only add results if syntactically similar code is found in at least 2 different locations. If the maximum value is set to 0, the CPP query Provider will add all results without a lower or upper bound cut off. Future version of C-CLEAR may replace the numeric fields with a slider control, as seen in Motive.

The fourth preference in the page allows query results to be serialised into a file, and requires a user to specify the location of the file generated by C-CLEAR. The file contains the query results in XML format, allowing other tools to use the query results. The results file is generated by the Generate Stats button in the Constructs View, and it contains the same exact information displayed in the Query Results View, along with additional generic, domain-independent information like the total number of constructs found in each file. These additional statistics are covered in greater detail in Chapter 4. This simple preference could be replaced by a browse dialogue in the future, allowing users to select a directory location in the file system. In addition, the results files will be automatically named based on the project selected by the user.

3.3.2 Domain and Tooling Functionality Separation

As with the Core layer, where domain definitions are separated from Syntax Model parsing and generation functionality, the Query layer also contains a clean separation of concerns.

Figure 29 shows the framework generically invoking a domain query and executing the query using common steps through a Syntax Model visitor and query task without any dependencies on any particular type of domain and without any knowledge of the query's purpose. The query's behaviour is contained entirely in the domain and defined by a domain implementation of `ICCLRQuery` (Figure 27). For the CPP domain, part of the implementation of this query is shown in Figure 30, where a Levenshtein analyser is used to compute syntactic similarities between conditionally compiled C code blocks. If the code is syntactically equivalent to other results, it is added to a `ResultsTable`.

Further separation of tooling framework and domain can be seen in the generic Query Preferences Page, where query parameters specified by a user can be interpreted by a domain query during the query evaluation, and allow for further adjustments to the query results. Delegating interpretation of these values to the domain allows for separation between tooling and domain. For the CPP query, the *Distance* query parameter is interpreted as a maximum Levenshtein distance for which two conditionally compiled blocks of code would be considered syntactically equivalent. Syntactically equivalent conditionally compiled code blocks are then added as query results.

The framework keeps track of general Syntax Model element information like location of the element in the source file, as well as generic information that the framework can compute about query results. Some of this generic information includes frequency of a selected construct in a Constructs View across files in the parsed software system, as well as number of times a query result is found in the code base. This information is then displayed to a user in the Query Results View. The fact that the

location of constructs and frequency of a results item is computed independent of any domain knowledge once again shows a separation between generic, reusable tooling functionality and domain. For example, C-CLEAR is not aware that constructs are actually CPP flags, or query results are conditionally compiled sections of code, but it still determines the location of the constructs, how frequently they were encountered, how many times a query result was found, and where in the source file the query result is located.

3.4 Summary

The Core**,** Query and UI functionalities in C-CLEAR are general and can be used to parse, model and display data from a domain without any particular hardcoded knowledge of that domain. It also enables certain facts of a software system to be marked as constructs that serve as a more meaningful context in which to invoke queries and analysis. This abstraction facilitates users in deciding what portions of a software system to analyse. Query results are also displayed in a common UI known as the Query Results View, and contain general statistics and location information about the query results independent of the domain. The domain query only needs to select elements in the Syntax Model as query results, and the framework will automatically keep track of the query results location in the source files. The framework's ability to track general properties of a Syntax Model and constructs independent of a domain allows for automatic integration into the IDE, as the query results location can be used to navigate from the Query Results View to the location in an IDE source file editor.

C-CLEAR maintains a clear separation between functionality and domain. Token definitions, grammar rules, construct marking, and query operations are all contributed

into C-CLEAR by a domain via framework extension points. C-CLEAR provides a common mechanism to generate data, display it, and perform operations on the information, but the interpretation of that data, and what actually gets displayed is the responsibility of a domain. Chapter 4 covers the evaluation of C-CLEAR on a number of different code bases.

Chapter 4: Evaluation

The motivation behind C-CLEAR focuses on two key features mentioned in Chapter 1: (1) establishing a clear separation of tool functionality from domain definition through a framework-defined API, such that focus is placed on a generic mechanism for extracting abstract concepts, and (2) automatic integration with an existing IDE.

Among the features that the C-CLEAR prototype aims to explore is generic functionality for marking constructs in source code as abstractions of interest to a domain. These abstractions are then presented in a common framework UI as selectable values which allow meaningful references to users when deciding what portions of a software system to query. A clear separation between common, reusable functionality and construct marking is achieved through an API. A domain decides what extracted data to mark as constructs through the API implementation, and C-CLEAR handles the indexing, management and display of the constructs.

The tooling framework has no semantic knowledge about the constructs beyond the fact that they are entities that need to be displayed to a user in a UI and that they are mapped to certain portions of the Syntax Model. It is up to the domain query to interpret these constructs during a query operation. The mapping of constructs to Syntax Model elements allow queries to focus on sections of the Syntax Model that would be of interest to the query. For example, the CPP Domain Language Provider marks CPP flags as constructs, and later uses selected flags from the Constructs View to evaluate their corresponding Syntax Model nodes. If the CPP directives associated with those nodes evaluate to true, the CPP query proceeds to analyse the conditionally compiled C code

using a Levenshtein algorithm. The intention of the query is to find syntactically similar conditionally compiled code for particular flag configurations.

The CPP domain tool that plugs into C-CLEAR only focuses on CCP-specific implementation, like CPP token definitions, grammar rules, construct marking, and Syntax Model querying. All actual tool functionality is cleanly separated into reusable tooling framework components. This includes selecting a set of projects containing the software system source code in an integrated IDE UI, parsing and modeling the source code into a Syntax Model, displaying marked constructs in the UI, executing a query and displaying query results in a Query Results View, and finally integrating navigation from a query results to other IDE components like source file editors.

As data is parsed and modeled into a Syntax Model, C-CLEAR keeps track of generic properties of extracted data, such as location information (file name and line number). This allows C-CLEAR to integrate with other IDE components independently of the parsed domain, and thus support navigation from query results to editor locations without having any semantic knowledge about the results or constructs.

Although the prototype CPP domain contribution into C-CLEAR was not meant to be a complete solution for addressing CPP usage in software configuration, it did serve to show that:

1. A tool developer can focus solely on domain logic to implement a tool that performs some software analysis task, such as searching for possible duplicate conditionally compiled code for certain configurations, using a tooling framework where all functionality and UI is already present. The separation of domain from tooling functionality is attained by a framework-defined API that the domain developer has to implement and contribute through framework extension points.
2. The tooling functionality automatically integrates with the IDE, therefore a domain developer will benefit from navigation to a source file editor from the tool's views. Since the extracted data is associated with file location properties, independent of the source type, automatic integration with the IDE allows users to navigate from query results to the corresponding line location in the source editor.

As C-CLEAR is only a prototype, the sample CPP domain is geared towards only analysing CPP directives that control conditional compilation. The CPP Language Provider marks CPP flags as constructs and C-CLEAR presents them in the Constructs View to aid users in deciding what to query in the source code. C-CLEAR itself has no knowledge that it is displaying CPP flags, only that the CPP domain has marked them as constructs to be shown in the Constructs View. It is up to a user to recognise abstract concepts from the displayed constructs, such as `LINUX32` and `LINUX64` flags indicating a Linux based configurations. The early version of C-CLEAR lacks any relationship support for the constructs, and therefore cannot leverage more meaningful visualisation that will assist a user to identify accurate configurations. However, the design of C-CLEAR would support the inclusion of this support in future versions.

In this evaluation of C-CLEAR, we focus on two key aspects of the framework design:

1. Whether the tooling framework's common data extraction and construct marking can indeed present meaningful abstraction to a user.
2. Whether automatic integration with the IDE was achieved by leveraging C-CLEAR's assumption that extracted data can always be associated with generic source location properties such as file name and line number independent of domain semantics.

In order to determine the efficacy of the prototype, C-CLEAR and the CPP domain Providers were evaluated against four open source code bases: Kaffe JVM [33], GNU's GCC [25], CVS 1.11.17 [15], and parts of OpenBSD [47] operating system. The first evaluation point is covered in Sections 4.1 and 4.2, determining if C-CLEAR's generic construct mechanism and Query framework can aid a user to better understand a highly configurable code base. Section 4.3 evaluates C-CLEAR's integration with the IDE.

4.1 Construct Metrics for CPP Domain

Chapter 3 overviewed the way in which C-CLEAR generates generic statistics on queried constructs based on common construct properties such as file location and frequency of construct occurrence in the code, independent of any domain semantics for those abstractions. In the CPP domain scenario, constructs are marked and interpreted by the domain as CPP flags. The frequency and locations of all selected constructs in the Constructs View as tracked generically by the framework are shown in Tables 4.1 and 4.2. In the context of CPP, the generic information gathered by C-CLEAR about constructs indicates significant CPP usage across files in the source code. Table 4 shows total number of flags and files containing those flags for GCC, OpenBSD's src/bin, CVS

and Kaffe JVM. C-CLEAR's generic construct mechanism indicates that GCC exhibits heavy usage of CPP directives which falls in line with what was shown by Ernst et all. [21].

Table 5 shows the breakdown of all the flags found in each source, grouped by how many flags occur in just one file, as well as 2-5, 5-10, and 10 or more files. For GCC C-CLEAR found 814 CPP flags that affect between 2-5 files each. As the prototype version of C-CLEAR used in this thesis does not support expansions of `#include`, files were calculated based on the actual source location of a CPP flag, as opposed to the usage of the flag through inclusion of dependant files containing the location of that flag. The number drops dramatically in the 5-10 files per flag range for all 4 sources, as seen in Figure 31, with only a modest drop in the 10 and over range. In most part, this appears to indicate that CPP flags for conditional compilation are contained largely within a few files for each flag, with a much smaller number affecting configuration over scattered locations across the source codes.

Source	**Total CPP directives flags**	**Total files containing CPP conditionals**
GCC	3526	1177
OpenBSD src/bin	94	84
Kaffe JVM	1026	501
CVS 1.11.17	614	170

Table 4: Total number of CPP flags and affected files.

Source	**Number of CPP flags affecting 1 file**	**Number of CPP flags affecting 2-5 files**	**Number of CPP flags affecting 5-10 files**	**Number of CPP flags affecting 10 or more files**
GCC	2599	814	76	37
OpenBSD src/bin	66	21	7	0
Kaffe JVM	607	382	20	17
CVS 1.11.17	445	144	19	6

Table 5: Breakdown of total CPP flags based on how many files contain each flag.

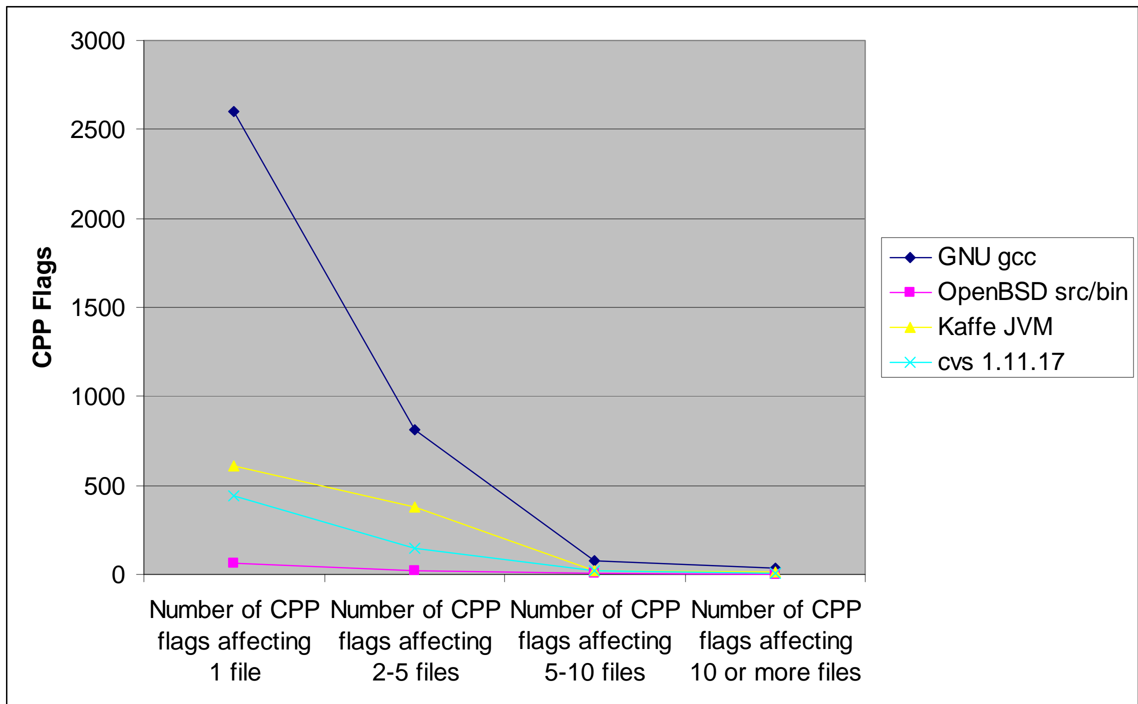

Figure 31: Graph displaying the CPP flag usage breakdown from Table 5.

The common properties of marked constructs can convey meaningful understanding of configuration within a software system without the framework itself having any semantic knowledge of what these constructs represent, as shown in the case of the CPP domain, where C-CLEAR's domain independent construct marking and indexing information was able to show CPP flag usage for conditional compilation across different source code.

4.2 Query Execution Results

Another part of the framework evaluation relies on analysing whether a domain query can utilise a common model structure like the Syntax Model and the abstracted constructs in the Constructs View in a query evaluation that generates meaningful data for domain analysis. Since the CPP domain marks CPP flags as constructs, this

evaluation will verify if selecting certain flags (i.e., configurations) and performing a Levenshtein algorithm on the controlled code blocks results in any useful data to the user. This preliminary evaluation shown in Figures 4.2 – 4.5 indicate that meaningful data can be obtained from a domain query evaluating nodes in the Syntax Model, even in the presence of a well defined separation of concerns between domain and tooling functionality—in particular the common, domain-independent abstraction mechanism and Syntax Model definition. Although the framework defines the Syntax Model elements, the structure is built by the domain Grammar Rules and later interpreted correctly by a separate domain query, showing that all domain-specific semantics about the Syntax Model and constructs reside entirely on the domain side of the tool.

The four test sources cover different types of software, from compilers (GCC) to version control (CVS) and JVMs (Kaffe). As it may not be possible to run queries on configurations that are common across the board, the Levenshtein-based CPP query is applied on all selected flags for each source, meaning that all conditionally compiled sections of code will be analysed for possible code duplication. Figures 4.2 – 4.5 show the CPP query executed on each of the code bases. The first column shows the actual results, in terms of blocks of syntactically similar conditionally compiled code.

With the exception of the Kaffe JVM, most results show scattered conditionally compiled configurations to be single lines of code, particular conditional compilation around `#define`, `#undef` and single function calls. Some configurations appear to inline additional conditions in a Boolean expression like `&& set -> extended,` as seen in Figure 32. Only those result entries that end in [..] indicate multiline conditionally compiled code, and the Kaffe JVM appears to have the largest number. The results show

that multiline conditionally compiled code blocks are not commonly duplicated for three of the code bases, but more prevalent for the JVM.

This analysis has shown that C-CLEAR's decoupled common tooling framework, construct management, and query execution can in fact show meaningful information on configuration code. Specifically, we were able to determine what type of configuration is more likely to be scattered across a source code base, and that the Kaffe JVM exhibits the most number of duplicate multiline conditionally compiled code blocks. In addition, if any of these conditionally compiled sections of code need to be changed, a tool like C-CLEAR may make it easier for developers to rapidly view all affected areas of a source and navigate to the locations and subsequently maintain the code using IDE editors.

Figure 32: Query execution on all flags for GCC.

C-CLEAR Constructs

Total Flag List Check All Clear All

Restore Previous Selection Query Selection

Generate Stats

- BACKWARDS
- BACKWARD_COMPATIBILITY
- BITS
- BRACE_EXPAND
- BSD
- BUFSIZ
- CONFIG_H
- DEBUG
- DEFAULT_ENV
- DES
- DMAXEXP
- EDEBUG
- EDIT
- EMACS
- EXTERN
- EXTERN_DEFINED
- FALSE
- FILEC
- FP
- HAVE_SOCKADDR_SA_LEN
- HISTORY
- IIASA
- INT_MAX
- JOBDEBUG
- JOBS
- KANJI
- KERBEROS
- KSH_DEBUG
- LINUX_TRANSLATE_H

C-CLEAR Query Results

PATTERN	TOTAL MATCHES	TOTAL FILES
etraci("exp0p1",p1,vp);	23	1
if(0	17	11
#definel__(i)=i	5	4
if(*fnp=='\0'&&*old_filename=='\0'){seterrmsg("nocurren [..]	3	1
Flag(FBRACEEXPAND)=1;	3	3
#undefEXTERN_DEFINED#undefEXTERN	3	3
i=S_ISFIFO(stb.st_mode);	3	1
if(des)init_des_cipher();	2	1
des=get_keyword();	2	1
if(check_addr_range(first_addr,current_addr+1)<0)	2	1
change_flag(FPOSIX,OF_SPECIAL,1);	2	1
{"braceexpand",0,OF_ANY},	2	1
{"monitor",'m',OF_ANY},	2	1
Flag(FEMACS)=Flag(FGMACS)=0;	2	2
FEMACS,FEMACSUSEMETA,	2	2
{XFUNC_stuff,0,CTRL('T')},	2	1
{"+fc",c_fc},	2	1
arcn->sb.st_size=(off_t)asc_ul(hd->c_filesize,sizeof(hd [..]	2	1
if(ul_asc((u_long)arcn->sb.st_size,hd->c_filesize,sizeo [..]	2	1
arcn->sb.st_size=(off_t)asc_ul(hd->size,sizeof(hd->size [..]	2	1
if(ul_oct((u_long)arcn->sb.st_size,hd->size,sizeof(hd-> [..]	2	1
STR_BSHELL=SAVE(_PATH_BSHELL);	2	1
(void)srcfile(_PATH_DOTCSHRC,0,0);	2	1
intk;(void)setlocale(LC_ALL,"");for(k=0200;k<=0377&&!Is [..]	2	1
i=S_IFIFO;gotofiletype;	2	1

Figure 33: Query execution on all flags for OpenBSD src/bin directory.

C-CLEAR Constructs

Total Flag List Check All Clear All

Restore Previous Selection Query Selection

Generate Stats

- HAVE_return_int
- HAVE_return_long
- HAVE_return_ref
- HAVE_returnarg_double
- HAVE_returnarg_float
- HAVE_returnarg_int
- HAVE_returnarg_long
- HAVE_returnarg_ref
- HAVE_sbc_int
- HAVE_set_label
- HAVE_set_lt_int
- HAVE_set_lt_int_const
- HAVE_spill_double
- HAVE_spill_float
- HAVE_spill_int
- HAVE_spill_long
- HAVE_spill_ref
- HAVE_store_addr_byte
- HAVE_store_addr_char
- HAVE_store_addr_double
- HAVE_store_addr_float
- HAVE_store_addr_int
- HAVE_store_addr_long
- HAVE_store_addr_ref
- HAVE_store_addr_short
- HAVE_store_byte
- HAVE_store_char
- HAVE_store_const_offset_byte
- HAVE_store_const_offset_int

C-CLEAR Query Results

PATTERN	TOTAL MATCHES	TOTAL FILES
(exc==0)&&	28	17
if(HAVE_add_int_const_rangecheck(val)){slot_slot_const([..]	26	2
lslot_lslot_slot(dst,0,src,HAVE_cvt_int_long,Tcomplex);	15	2
voidadc_int(SlotInfo*dst,SlotInfo*src,SlotInfo*src2){sl [..]	14	2
if(HAVE_load_offset_int_rangecheck(offset)){slot_slot_c [..]	14	2
if(HAVE_store_offset_int_rangecheck(offset)){slot_slot_ [..]	14	2
mallopt(M_MMAP_MAX,0);	12	12
if(reginfo[s->regno].ctype&Rlong){spill_long(s);}else	11	2
lslot_slot_lslot(0,dst,src,HAVE_store_long,Tstore);	10	2
slot_slot_slot(dst,NULL,src,HAVE_move_float,Tcopy);	9	2
{SlotInfo*tmp;slot_alloctmp(tmp);move_int_const(tmp,val [..]	9	1
_add_int(dst,src,src2);}	9	1
slot_slot_slot_const_const(dst,src,idx,offset,0,HAVE_lo [..]	8	1
slot_slot_slot_const_const(dst,idx,src,offset,0,HAVE_st [..]	8	1
sigaddset(&newact.sa_mask,SIGVTALRM);	8	4
cachehits++;	7	4
slot_slot_const(dst,0,(jword)l,HAVE_load_constpool_int, [..]	7	1
slot_slot_const(dst,NULL,(uintp)addr,HAVE_load_addr_int [..]	7	1
slot_slot_const(NULL,src,(uintp)addr,HAVE_store_addr_in [..]	7	1
voidreturn_int(SlotInfo*dst){slot_slot_slot(dst,NULL,NU [..]	7	2
voidreturnarg_int(SlotInfo*src){slot_slot_slot(NULL,NUL [..]	7	2
#definePROM_ij	6	5
lslot_lslot_lslot(dst,0,src,HAVE_move_anylong,Tcopy);	6	2
void_sub_int(SlotInfo*dst,SlotInfo*src,SlotInfo*src2){s [..]	6	1
slot_slot_slot(dst,NULL,src,HAVE_neg_int,Tcomplex);	6	2
voidload_int(SlotInfo*dst,SlotInfo*src){slot_slot_slot([..]	6	2
if(reginfo[s->regno].ctype&Rref){reload_ref(s);}else	5	2
gotofound;}	5	5
if(slot_type(src)==Tconst){{add_int_const(dst,src2,slot [..]	5	1
if(slot_type(src2)==Tconst){sub_int_const(dst,src,slot_ [..]	5	1
voidstore_int(SlotInfo*dst,SlotInfo*src){slot_slot_slot [..]	5	2
slot_slot_lslot(dst,NULL,src,HAVE_cvt_double_int,Tcompl [..]	5	2

Figure 34: Query execution on all flags for the Kaffe JVM.

Figure 35: Query execution on all flags for CVS.

4.3 IDE Integration

The final evaluation condition tests C-CLEAR's integration with the IDE. As C-CLEAR is based on file parsing, certain common features will be present regardless of domain, like the source files in the IDE workspace, and location of constructs and queried data in those files. Therefore navigation can be implemented generically enough such that double clicking on any entry in the Query Results View will open the file at the correct location, and even highlight the construct that was used in the query, as seen in Figures 4.6-4.9, without the framework having any additional knowledge of what these abstracted constructs are or what the query results indicate.

When expanding any particular query result, the lowest nodes are line numbers within a file that indicate where the result was found. For GCC, double clicking on the "642" entry will open the C editor for *cfglayoud.c* and highlight the CPP flag that was

evaluated in the query, which in this case is ENABLE_CHECKING. Note that the line number indicates the location of the actual result as opposed to the construct that was used to evaluate the query. A similar navigation from the Query Results View to the C editor can be seen for OpenBSD, Kaffe and CVS in Figures 4.7, 4.8, and 4.9, respectively.

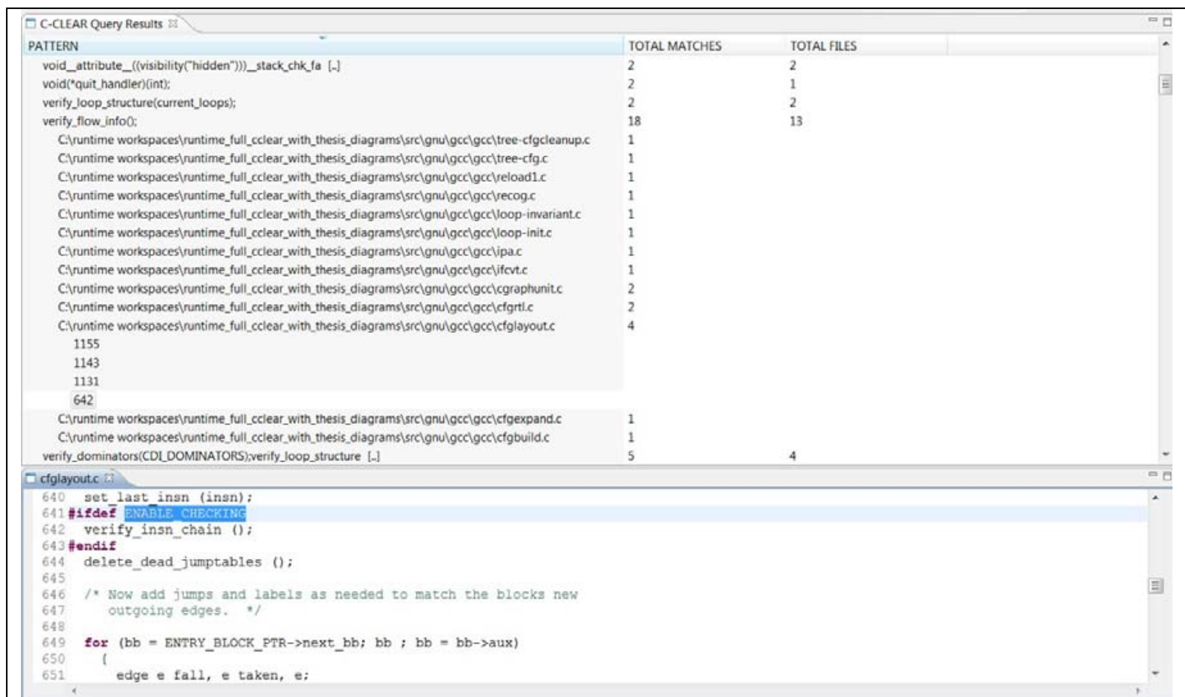

Figure 36: Navigation from a query result to the C editor for a file in GCC.

C-CLEAR Query Results

PATTERN	TOTAL MATCHES	TOTAL FILES
etraci("exp0p1",p1,vp);	23	1
if(0	17	11
#defineI_(i)=i	5	4
if(*fnp=='\0'&&*old_filename=='\0'){seterrmsg("nocurren [..]	3	1
C:\runtime workspaces\runtime_full_cclear_with_thesis_diagrams\src\bin\ed\main.c	3	
543		
702		
842		
Flag(FBRACEEXPAND)=1;	3	3

main.c

```
540         if (*fnp && *fnp != '!')
541             strlcpy(old_filename, fnp, sizeof old_filename);
542 #ifdef BACKWARDS
543         if (*fnp == '\0' && *old_filename == '\0') {
544             seterrmsg("no current filename");
545             return ERR;
546         }
547 #endif
548         if (read_file(*fnp ? fnp : old_filename, 0) < 0)
549             return ERR;
550         clear_undo_stack();
551         modified = 0;
552         u_current_addr = u_addr_last = -1;
553         break;
554     case 'f':
555         if (addr_cnt > 0) {
556             seterrmsg("unexpected address");
557             return ERR;
558         } else if (!isspace(*ibufp)) {
559             seterrmsg("unexpected command suffix");
560             return ERR;
561         } else if ((fnp = get_filename()) == NULL)
562             return ERR;
563         else if (*fnp == '!') {
564             seterrmsg("invalid redirection");
565             return ERR;
566         }
```

Figure 37: Navigation from a query result to the C editor for a file in OpenBSD.

C-CLEAR Query Results

PATTERN	TOTAL MATCHES	TOTAL FILES
(exc==0)&&	28	17
if(HAVE_add_int_const_rangecheck(val)){slot_slot_const([..]	26	2
C:\runtime workspaces\runtime_full_cclear_with_thesis_diagrams\kaffe\kaffe-1.1.9\kaffe\kaffevm\jit3\icode.c	13	
980		
1023		
1166		
1327		
1513		
1607		
1810		
1885		
1945		
1997		
2074		
2133		
3999		

icode.c

```
1162 void
1163 sub_int_const(SlotInfo* dst, SlotInfo* src, jint val)
1164 {
1165 #if defined(HAVE_sub_int_const)
1166     if (HAVE_sub_int_const_rangecheck(val)) {
1167         slot_slot_const(dst, src, val, HAVE_sub_int_const, Tcomplex);
1168     }
1169     else
1170 #endif
1171     {
1172         SlotInfo* tmp;
1173         slot_alloctmp(tmp);
1174         move_int_const(tmp, val);
1175         _sub_int(dst, src, tmp);
1176         slot_freetmp(tmp);
1177     }
1178 }
```

Figure 38: Navigation from a query result to the C editor for a file in the Kaffe JVM.

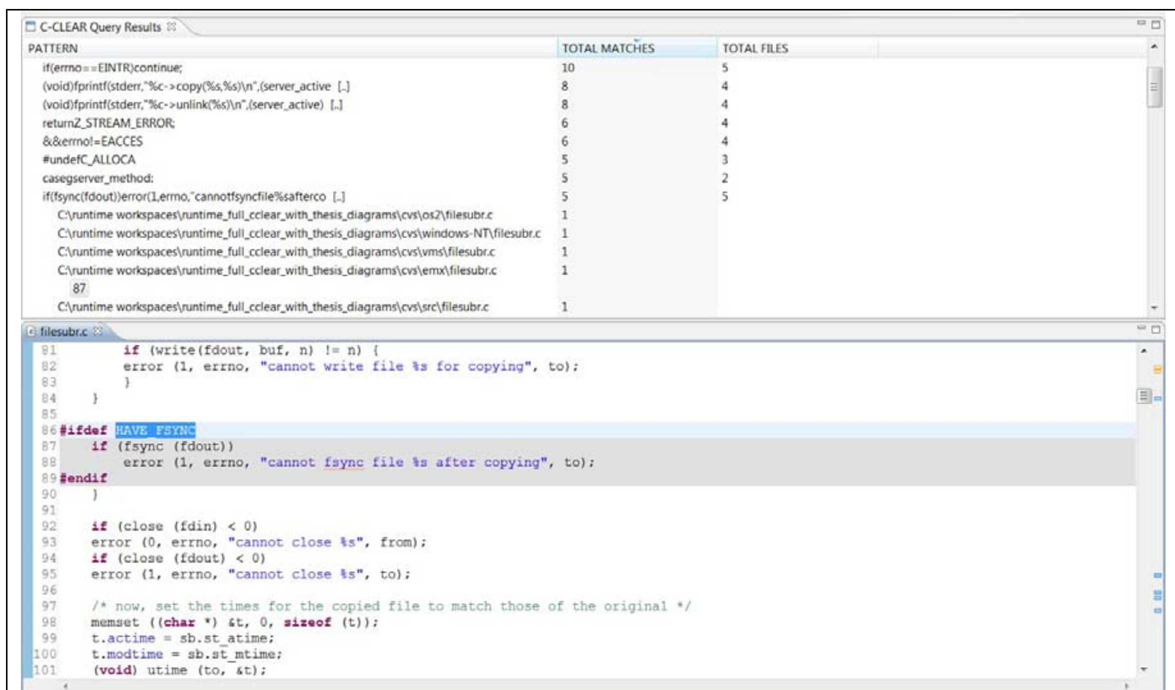

Figure 39: Navigation from a query result to the C editor for a file in CVS.

4.4 Framework Evaluation for a Comment Analyser

As C-CLEAR is intended to be a general, extensible framework for software analysis tool development, to further evaluate the reusability of the framework for other domains aside from CPP, this section will present a simpler evaluation of the framework geared towards finding syntactically similar comment blocks in some high level language. This proof-of-concept study will generally describe the steps needed to implement an analysis tool that identifies similar comments within a code base leveraging the capabilities of the C-CLEAR framework.

There are three types of information that the comment analyzing tool would have to provide to C-CLEAR: 1) a list of tokens to extract from the source, 2) language rules for building the intermediate representation of the comment blocks within a code base, 3)

a query that performs analysis on the intermediate model elements representing comment blocks.

The C language defines comments as blocks of text in the source preceded by either “//”, or starting with a “/*” and ending with a “*/”, where the content within the block starts with a “*”. Therefore the comment language provider forming part of the comment analysis tool contributed into the C-CLEAR framework would define the following four tokens: “//”, “/*”, “*”, “*/”.

The second step is to define a language rule that builds the intermediate syntax model. It is the job of the comment language provider to build the model based on the stream of tokens processed by the underlying C-CLEAR framework. As the pattern for comments is fairly straight forward, the provider would detect within the token stream provided by C-CLEAR the first two tokens: the comment symbol and the comment line following. It therefore builds a model element representing a comment block by detecting the starting and ending comment token for the “/*” and “*/” case, or in the case of “//”, by detecting a token that is not preceded by a “//” token.

The provider also has the option of marking constructs to display in the Constructs View that can facilitate querying and analysis of comments in the source. In this case, the comment provider creates construct objects representing files containing comment blocks, and marks them as selectable values using C-CLEAR’s API that the framework then displays in the Constructs View.

The comment analysis tool would also contribute a query into C-CLEAR that would identify a comment model element as C-CLEAR visits the intermediate model during a query operation. If the comment is identified, the comment analysis tool query

can use one of C-CLEAR's general, built-in analysers available through a factory to detect syntactically similar code blocks. Both the CPP and comment Query providers use a C-CLEAR Levenshtein comparator that is applicable to both cases, as the comparator only requires string representation of the model element values (either the raw conditionally compiled code, or the comment body text).

An extension to this implementation of a comment-specific analysis tool contributed into the C-CLEAR framework, comments for other programming languages can be taken into account. For example, to include Javadoc the comment model provider would have to define a fifth comment token type "/**". The language rule would remain the same, with the exception of adding a model element indicating a Javadoc block.

Additionally, extending the identification of constructs to be used by the Java comment query provider, with two types of comment blocks, Javadoc and inline comments, the specialised comment analysis tool can mark the block type as a construct itself. This introduces a level of abstraction that can be leveraged at the C-CLEAR UI layer allowing a user to narrow a search within one of the two.

4.5 Conclusion

C-CLEAR is a framework that separates functionality and UI into a common set of components that tool developers can use to build tooling for software configuration analysis using the same resources and work area of an IDE. The framework was designed such that portions of a complex software system can be analysed and code maintained through the use of common views and automatic integration into the IDE, especially the file editors. However, there are no dependencies on any particular domain in the framework, and therefore C-CLEAR can theoretically be used on a number of different

Domains where configuration of a software system is specified in files, whether at source code level using either pre-processor directives or comments, build scripts or XML. Basically, as long as the information is of interest to a user, we may be able to support its capture and representation through a common series of steps.

Evaluation of the tooling framework along with the sample CPP domain contribution has shown that generic statistical information collected by C-CLEAR, such as the location and frequency of construct usage, can present meaningful insight into a software system that uses configuration to support fine-grained code-level variations. Analysis of four different code bases showed the frequency of CPP flag usage for conditional compilation, and indicated that in most cases, CPP usage per flag is contained within a few files. Only certain flags are seen to affect more than 10 files per code base.

In addition, C-CLEAR was able to reveal the type of configuration code that is most commonly scattered across a source base, with single line code for `#define` and `#undef` being among the most common. Allowing a domain to mark concepts of interest through an API decouples any semantic knowledge about those constructs from C-CLEAR, containing them entirely in domain implementations as well as user cognition when browsing the list of constructs in the Constructs View.

Finally, generic location information about constructs and queried data such as file name and line number location for both the constructs and the query results shows that integration with other IDE components can be automatically leveraged by a domain tool developer. Essentially, this means the developer does not need to be concerned with determining these location properties for the extracted data, as that is done by C-CLEAR during the Lexical Tokenising phase. Navigation and highlighting of constructs and query

results are automatically performed by the C-CLEAR framework with no additional semantic knowledge of these constructs, what type of source was parsed, or what the query results represent.

Chapter 5: Conclusion and Future Work

C-CLEAR is a first pass look at an integrated tooling framework design for configuration management, where separation of concerns between tooling functionality and domain-specific logic is clearly established using an API and extension points. One of the primary focuses of the tooling framework is to determine the type of information that is required from a domain. Chapters 2 and 3 established that the four main types of information needed from a domain are:

1. Token Definitions.
2. Grammar Rules that decides the structure of a common Syntax Model.
3. Marked facts of the extracted data that represent abstract concepts meaningful to a tool user, and serve as references for executing queries.
4. Queries that understand the Syntax Model structure and abstract constructs, and perform an analysis operation.

On the tooling framework side, the minimum common reusable components for parsing, modeling, displaying and querying a software system are:

1. A lexical tokeniser, with a hook for domain Token Definitions.
2. A syntax parser, with a hook for domain Grammar Rules.
3. Generic construct marking mechanism that requests marked data from a domain using the same hook as the Grammar Rules.
4. IDE-integrated UI for selecting a set of source projects from the IDE's workspace.
5. A view for displaying constructs that has the same look and feel as other IDE views, and where queries can be launched based on selected constructs.
6. A querying framework that uses selected constructs from the Constructs View, loads a domain query from a framework hook, and manages the query results.
7. A view for displaying query results, with similar look and feel as other IDE views, where general, domain-independent properties like file name and line number location for the query results are shown.
8. IDE integration with the Query Results View where navigation to an IDE source editor from a query results entry is possible, regardless of the parsed domain. The navigation also highlights any constructs it can identify near the query results in the editor, as seen in Chapter 4.

Allowing the domain to define the language specifics of a software system, as well as determine any abstractions in the extracted data, and establishing queries that understand the language specifics and abstractions cleanly separates common tooling functionality from domain implementation. This allows a tool developer to leverage C-CLEAR for all functionality, IDE integration, and UI, and focus entirely on domain logic.

In order to validate this design, a sample domain, CPP, was implemented to evaluate the separation of integrated tooling functionality and domain definitions. Chapter 3 demonstrated how a tool developer needed to only focus on CCP-based implementations, such as defining CPP token types and CPP syntax rules for conditional compilation, as well as marking CPP flags as constructs of interest to a CPP query. It also defined a query which understands the marked constructs as CPP flags as well as Syntax Model nodes that represent CPP conditional compiled sections, and can therefore evaluate those nodes using selected constructs. The evaluated nodes were then compared against each other for syntax similarity using a Levenshtein algorithm.

Chapter 4 showed that this separation between C-CLEAR and the CPP domain was able to generate meaningful data about a C-based software system, including: frequency of CPP flag usage for conditional compilation, types of syntactically similar conditionally compiled code that are commonly scattered across source files, and whether certain software systems feature more multiline code duplication than other types. All this was accomplished without the C-CLEAR framework having any dependencies on CPP. The domain information that C-CLEAR needed to generate the CPP data and query results was obtained entirely using well defined extension points and API.

In addition, C-CLEAR assumes that all extracted data and abstract concepts can be mapped to source locations independent of the domain. Therefore, C-CLEAR keeps track of file and line number locations of all extracted data and constructs. This allows the framework to implement navigation from a query results in the Query Results View to a source location in an IDE editor without having any semantic knowledge of what the query results or constructs represent.

The separation between tooling functionality and domain worked satisfactorily for the simplified case of the CPP domain, where focus was entirely on CCP conditional compilation and CPP flags that control code inclusion. However, limitations in the C-CLEAR API prevent the CPP domain from establishing relationships between constructs. Additionally, evaluation of C-CLEAR was performed on only one type of domain. Whether C-CLEAR's Syntax Model and construct marking API can support other domains remains to be seen. Future work will also address construct relationship support at the API level and corresponding visualisation at the C-CLEAR UI layer. Sections 5.1, 5.2, and 5.3 take a closer look at concrete proposals for future work that build upon this thesis.

5.1 Supporting Construct Relationships

Visualisation is one of the primary focus areas for future C-CLEAR enhancements. For the CPP domain, the C-CLEAR framework managed to generate meaningful data without any dependency on CPP. However, the CPP case was relatively straightforward, focusing only on conditional compilation and overlooking CPP flag dependencies. Consequently, accurate configurations for a given C-based source file are not visualised using C-CLEAR's Constructs View, and configuration selection is

manually set by a user's knowledge of the system. This greatly limits C-CLEAR in terms of abstracting, visualising and managing actual build configuration for a C source, in part because relationships between configuration constructs like CPP flags are not yet supported. Ideally, the Constructs View would be replaced by a more complex visualisation where groups of flags or constructs represent an abstraction that allows a user to better understand the architecture, configurations and dependencies of a software system.

Chapter 4 showed that some conditionally compiled code sections are just bracketing further one-line CPP flag definitions for `#define`. As the CPP domain defines the Grammar Rules for building a Syntax Model, it should be possible to establish relationships amongst CPP flags while parsing CPP directives. If conditionally compiled code includes `#define` statements, a relationship can be established between the flags that control the inclusion of these statements and the flags defined by the inclusion. However, the C-CLEAR API needs to support any type of relationship between constructs, regardless of the domain. Consequently, determining a generic API and relationship model that would allow different Domains to establish construct relationships is important future work for C-CLEAR.

5.2 Proposed Query Results Visualisation

In addition to the Query Results View, future work may consider other forms of visualisation for displaying query results like diagrams. Emphasis will be placed on whether graphical visualisation may scale to large number of results, or an enhanced version of the current table form in the Query Results View would be more adequate. In particular, if the current table is enhanced, colour coding and highlighting of results can

be used to aid comprehension. Related constructs that represent an actual configuration can assist developers in determining what sections of a code base are affected by the configuration, rather than what files are affected per individual flag. The Query Results View only shows locations for syntactically equivalent conditionally compiled code in a table, as seen in Figure 32 – 4.5, but does not show which constructs are responsible for each result, and if there are any relationships between the constructs. Future C-CLEAR work will complement the Query Results View with an additional visualisation diagram that better shows which files are affected by the selected configuration flags.

Some preliminary prototype work for enhancing C-CLEAR to support construct relationships and results visualisation is shown in Figure 40, where a GMF-based [26] diagram similar to what is used in IBM's RAD [28] was used to automatically visualise the results of a CPP query operation performed on three selected flags: `EXTENDED_BUFFER`, `LINUX`, and `WIN32`. The diagram visualises the impacted files that contains the queried CPP flags on the right side, and the selected CPP flags used in the query on the left side, with connectors between the two groups. Different colours are used to indentify the CPP flags, and each affected file contains a nested colour-matched flag element indicating the actual conditionally-compiled code section controlled by that flag. In this early prototype, double clicking on any of the nested flag views inside the enclosing file elements would navigate directly to the source editor location for that code.

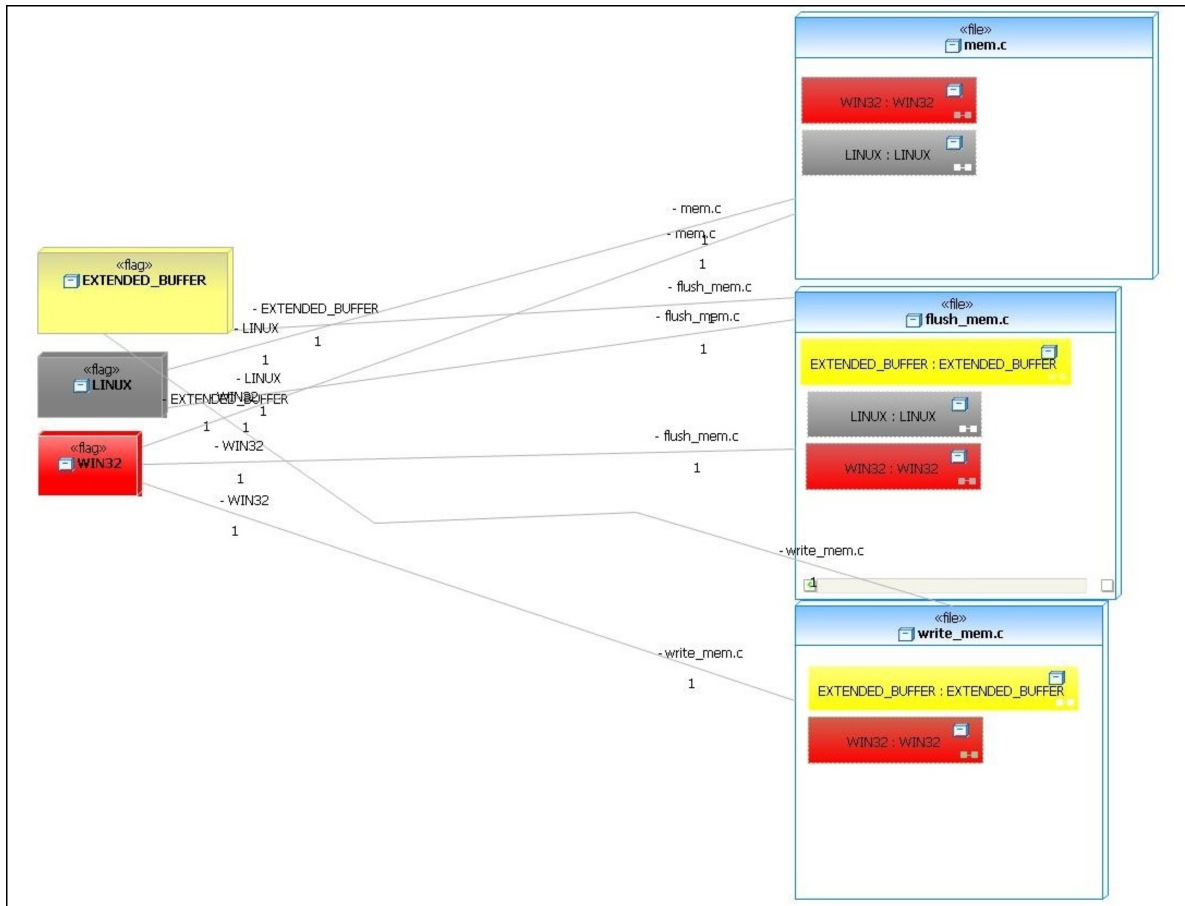

Figure 40: Prototype visualisation for C-CLEAR.

Clearly, for files containing many affected regions, this visualisation may not scale well if each affected area of the code is represented by one nested flag view, as a file may contain hundreds of affected areas.

A possible solution is to have at least two visualisation levels in the diagram, where the topmost level simply shows all the selected flags on the left and all the files affected by those flags on the right, as in Figure 40, but the nested flag elements inside the file views do not actually indicate code locations. That means that each file view in the first level mode would have at most one distinct nested flag element, indicating that it contains results related to that flag.

Double clicking on a flag element on either the left side list in Figure 40, or one of the nested flags in the file views on the right side will expand the diagram to a second-level mode. The second-level mode of the diagram still retains all the flags listed on the left side of Figure 40, but one or more of the file views on the right side are each replaced with a view similar to *make.def.template* in Figure 41, depending on three different actions listed in Table 6. This deeper level view of a file will be referred to as a "second-level" view. If a single flag is selected on the left side of the diagram, all affected files views on the right side are replaced with second-level file views. Only those file views that are connected to the selected left hand side flag are shown, and all other ones are hidden. Likewise, a multiple selection of flags on the left side will result in a similar state: all affected files and their connectors are displayed in second-level view mode, and anything else is hidden. Finally, selecting a nested flag element in a file view on the right side in Figure 40 will only expand that file into second-level mode. All other files views remain unchanged in first-level view mode, as well as the flag list on the left side, except that the only connector shown is the connector from the nested flag element back to the corresponding flag on the left side.

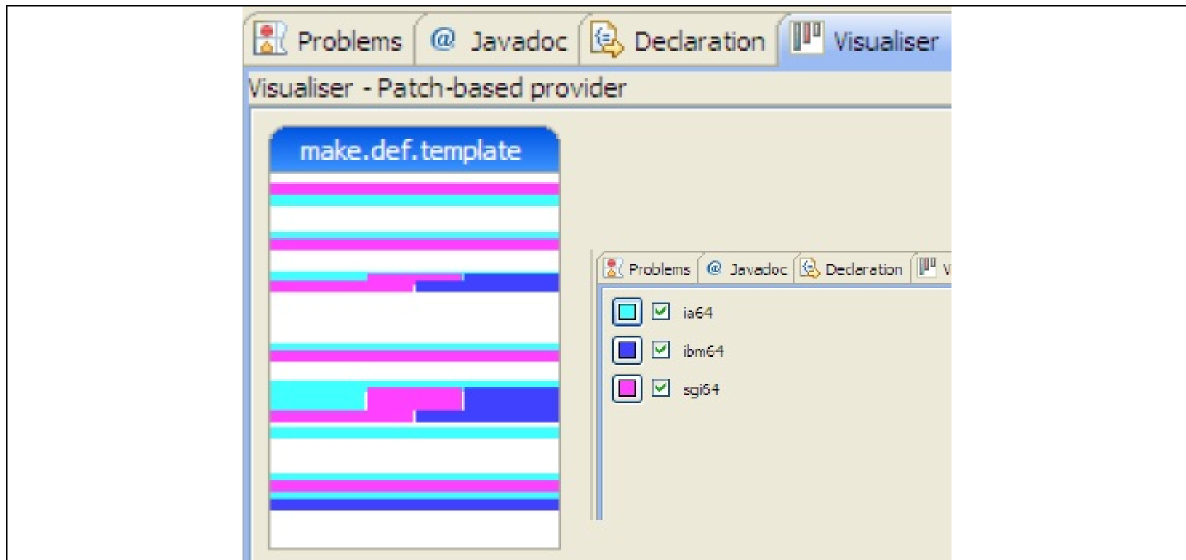

Figure 41: Proposed visualisation where horizontal blocks indicate code areas affected by a configuration construct.

Figure 41 shows a visualisation technique proposed originally in a tool called Déjà View [14], where a vertical view represents a file, and horizontal areas inside the view indicate configuration code locations as they appear in the actual source file, where different colours pertain to separate configurations. The proposed second-level file view would be similar to Figure 41, and would have the option of being sorted. In the second-level mode, the file view would resemble the *make.def.template* in Figure 41, where the horizontal blocks indicate places in the code affected by a selected flag.

A user would also be able to sort the second-level view of a file based on query result locations that appear only once, versus query locations that are duplicate or similar across the file. As each flag is colour-coded uniquely, the flag colours will be graded based on that colour within a second-level view. The base colour would indicate single, non-duplicate results, and increasing gradients would indicate locations that contain duplicate results, with the highest gradient being the most duplicated result. In addition, selecting any of the horizontal blocks in the second-level view will highlight all other

locations in the file view, or even across visible file views, that contain that same result, as well as highlighting the file figures themselves. This would allow a user to quickly determine which files contain the selected result, and because the results are colour-coded and connected to a selected flag on the left hand side, it would be easy to determine which construct generated that query result.

Selection in the Figure 40 Diagram	Resulting Action
Select and double click on a flag in the flag list on the left side of the diagram.	All affected file views on the right side of the diagram are replaced by a second-level, vertical view similar to *make.def.template* in Figure 41. Only the relationship connectors between the selected left side flag and all affected second-level file views are shown. The rest are hidden.
Select and double click on multiple flags in the flag list on the left side of the diagram.	Same as above.
Select a nested flag in any of the first-level file views on the right side.	Only the file view containing the selected flag is replaced by a second-level version of the file view. Only the connector from that file to the corresponding flag on the left side is shown, and all others are hidden.

Table 6: Proposed behaviour of Query Results Visualisation.

All the proposed visualisation and data interactions are listed in Figure 42. Other interactions will include navigating the source file location by double clicking on any of the horizontal blocks in the second-level file view. Hovering over a block will show the result text in a popup window. Finally the query results diagram will be automatically synchronised with the files, so changes in the files will reflect on the diagram. Likewise, users will be able to delete horizontal blocks from the second-level view, and corresponding changes in the source file will be automatically performed by C-CLEAR. Users will have the option to localise the changes to just that one file, or across multi-selected file views, or all affected file views that contain that same query result.

1. Height of horizontal block determines the number of actual lines in the source file containing the query result.
2. Graded colour indicates query result duplication level, with base colours indicating single occurrences of the result, and graded shadings higher levels of duplication.
3. Selecting a particular horizontal block will highlight all the locations across all visible second-level file views that contain that same exact result. The affected file views will also be highlighted. That way a user can quickly find all affected areas across different files that contain that particular result.
4. Double clicking on any of the horizontal blocks will open the IDE source editor for that file type and navigate to the location containing the result.
5. Each or all second-level file views can be sorted based on query result duplication. By default, the horizontal blocks appear in the order that the corresponding query results appear in the source file.
6. Hovering over each horizontal block will display the corresponding query result in a popup window.
7. Ideally, file operations can be performed right from the second-level view. If a certain selected horizontal block is deleted, the corresponding source level text is removed as well. Likewise, changes in a file editor will be automatically reflected in the query results diagram without having to run the query again. The user would have the option of performing the action on just one file, selected files, or all affected files.

Figure 42: Proposed visualisation and interaction properties in a second-level file view.

As with all other components of C-CLEAR, the proposed visualisation of query results will be generic and domain independent. The CPP domain will be used to evaluate the visualisation, similar to what was done in Figure 40. Further studies on other Domains will determine if this visualisation will also work to show relationships between constructs and query results, and whether the second-level view operations like deleting horizontal code blocks are sound and yield correct results across different Domains. The CPP domain itself will be enhanced to support CPP flag dependencies, particularly where CPP flags control the inclusion of other flags, either using `#define` or including header files.

The proposed visualisation would also have to be adjusted to support construct relationships. Instead of listing the flags individually on the left side of the diagram as in Figure 40, the flags (or constructs for other Domains) may be clustered by common relationship. That is, all related flags would be grouped into one visualised element, and relationship connectors would connect to all affected files to that cluster or group. Also,

the connectors should be colour coded to match the source end point, to allow for better visual tracing. The visualised element representing a group or cluster of flags will also be at least two levels deep, the first level showing the group as one concept, and the second level showing all the containing flags, including all the individual connectors from each flag connecting to the affected file views on the right side of the visualisation.

5.3 Framework Enhancements

Chapters 2 and 3 mentioned various proposals for enhancements to C-CLEAR in the context of the existing prototype. One limitation mentioned is that a domain can only be defined by one Language Provider. This means that when an input is parsed and modeled, only one Language Provider is used for the entire session. If the source code contains different Domains, like C (in other words, a ".c" or ".h" file), makefiles, and XML, or even Java or Javascript, C-CLEAR will only load the Language Provider for the first file type that has a registered domain. So even when there are C, makefile, XML, Java and Javascript Domains registered into C-CLEAR, if the first encountered file is a C file, only the C Language and Query Providers will be used throughout the entire parsing and querying processing. Future work may address support for multiple Domains within the same parsing session, and even look into merging Syntax Models from each domain into one structure. Most likely, one of the Domains would establish Grammar Rules that would handle cross-domain merging. As this is a very open ended and potentially vast area of research, future versions of C-CLEAR will most likely only support limited merging, or require a domain to be registered against different file types and avoid any merging scenario. For example, if a tool developer wishes to model data across .java and .xml files, the developer would implement one domain registered against both of these,

instead of having one domain for each and determining some complex merging logic in one or both of the domain's Grammar Rules.

Additionally, Figure 23 in Chapter 3 shows C-CLEAR being invoked from one context menu, and will load either the domain with the highest priority for an encountered file type in the selected projects, or the first domain for providers of equal priority. Ideally, C-CLEAR would replace the context menu action with a menu group, where different Domains can be selected, or launch a C-CLEAR dialogue or wizard from a single context menu action where a list of different Domains for the selected projects is available.

Also mentioned in Chapter 2 is C-CLEAR's support for only one query per domain. Ideally, a domain Query Provider would contribute multiple queries and these would be available for selection in a drop-down menu from the Constructs View (Figure 24) instead of a Query Selection button. The Restore Previous Selection button would also be replaced with a history drop-down menu, allowing backward selection of n number of previous queries. This would avoid the user from having to reselect constructs in the view when recreating an older query result.

Another area that will require further analysis is whether the generic query parameters in the Query Preferences Page (Figure 26) can be used to adjust query results for other Domains aside from CPP. The *Deviation* parameter was successfully used by the CPP query as an upper bound Levenshtein distance, where comparisons that fall within the *Deviation* are considered syntactically similar. How other Domains may interpret the *Deviation* parameter remains to be studied. Replacing the numeric fields for the *Deviation* and maximum and minimum parameters with slider controls will also be

looked into, in particular if a domain defines fixed bound values for these three parameters. If no bounds are present, the numeric fields as seen in Figure 26 would be sufficient.

C-CLEAR also allows for query results to be serialised into files. In the prototype version, a file with the same name is generated for every query, and existing files are overwritten. Ideally, this should be replaced with an automatic naming mechanism that creates new files for each query, and automatically names them according to the query type, domain name, and parsed project. If executing the same query repeatedly, a similar file name would be used as the previous queries, but with automatic incremental number, like for example: *kaffe_CPP_levenshtein1.xml*, *kaffe_CPP_levenshtein2.xm*, where the underscore separates project name, domain type, and query name. These would be automatically be generated by C-CLEAR based on the registered properties of a domain and the project selection. The location text field where files should be generated will also be replaced with a file system browsing dialogue, to allow users to select a folder location for the generated files.

Another minor change to C-CLEAR includes API modifications to the query interface shown in Figure 27, where the prototype version relies on a descriptor argument to store query results. A better API would be to return the query result to C-CLEAR.

The query framework would also have to be adjusted when construct relationship support is introduced. If a construct relationship is modeled with a type property, queries may potentially be configurable at the UI level, meaning that the proposed drop-down menu in the Constructs View that lists all available queries may invoke a query configuration dialogue or wizard prior to launching a query, with the option of

remembering a preference stored in the Query Preference Page for a particular query. For example, in a future version of the CPP domain Query Provider, a query may support finding all flags that were *undefined* using `#undef` by a selected set of CPP flags. The *undefined* relationship would be a relationship type defined by the CPP Language Provider during the construct marking phase, and would appear on a query UI as a selectable relationship to query.

5.4 Conclusion

C-CLEAR's design and implementation showed that tooling functionality and domain definitions can be cleanly separated using an API. Various components on either side of the tool were defined, with the domain contributions provided through framework extension points. For evaluation a sample CPP domain was implemented to parse CPP conditional compilation directives. The implementation only contained CPP logic and no further tooling code was needed to integrated into C-CLEAR and Eclipse. IDE integration was performed automatically by C-CLEAR, in particular navigation from the Query Results View to source file editor.

Evaluation of the CPP domain and C-CLEAR framework showed that meaningful data about a C-based source code can be obtained without the framework having any knowledge or dependencies on CPP or any other domain. Some of the evaluation showed that CPP usage for conditional compilation across different software systems feature infrequent, multiline code duplication for most systems in the study, except the Kaffe JVM. The generic properties of extracted data that C-CLEAR keeps track of independent of the parsed domain such as file name and line number location were able to show CPP flag usage across files in a source code, with most flags affecting only a few files.

The CPP domain implementation has so far indicated a clean separation between tooling functionality and domain definition for tooling geared towards software configuration analysis. Future work will focus on other Domains, such as build scripts and XML-based software configuration. This evaluation can also determine if we can leverage C-CLEAR without any additional functional tooling implementation, or whether the API and common Syntax Model established by C-CLEAR require modification. Particular emphasis will be placed on stronger visualisation solutions for query results that show relationships between constructs and results, as well as between the constructs themselves. The generic construct marking mechanism will also be enhanced to support construct dependencies and relationships, and higher level abstractions based on these relationships.

Although research with other Domains will verify the robustness and reusability of C-CLEAR as an integrated, reusable tooling framework, the CPP domain implementation used to evaluate the separation of framework functionality and IDE integration from domain-specific logic demonstrated that the C-CLEAR framework succeeded in establishing a clear division between the two, and that tooling design is a valuable area of research that may benefit from more flexible approaches along these lines.

Bibliography

[1] A. Telea and and O. Ersoy and L. Voinea: **Visual Analytics in Software Maintenance: Challenges and Opportunities**, Proc. 1st European Symposium on Visual Analytics, 2010.

[2] A. Telea, L. Voinea: **An Interactive Reverse Engineering Environment for Large-Scale C++ Code**, SoftVis '08 Proceedings of the 4th ACM symposium on Software visualization, 2008.

[3] Ahmed E. Hassan and Richard C. Holt: **C-REX: An Evolutionary Code Extractor for C**, 2004.

[4] Ahmed E. Hassan and Richard C. Holt: **Replaying development history to assess the effectiveness of change propagation tools**, Empirical Software Engineering Volume 11 Issue 3, September 2006.

[5] Ahmed E. Hassan and Richard C. Holt: **Using Development History Sticky Notes to Understand Software Architecture**, 12th IEEE International Workshop on Program Comprehension, 2004.

[6] Algorithms and Theory of Computation Handbook, CRC Press LLC, 1999, "**Levenshtein Distance**", in Dictionary of Algorithms and Data Structures, Paul E. Black, ed., U.S. National Institute of Standards and Technology. 10 November 2005.

[7] Andrew McNair, Daniel M. German, and Jens Weber-Jahnke: **Visualizing Software Architecture Evolution using Change-sets**, WCRE '07 Proceedings of the 14th Working Conference on Reverse Engineering, 2007.

[8] **Apache Harmony**: http://incubator.apache.org/harmony/, April 2011.

[9] Badros, G. J., and Notkin, D: **A framework for preprocessor-aware C source code analyses**, Software - Practice and Experience 30, 8, 907–924, 2000.

[10] Bram Adams, Kris De Schutter, Herman Tromp and Wolfgang De Meuter: **The Evolution of the Linux Build System**, SPLC'10 Proceedings of the 14th international conference on Software product lines, 2010.

[11] Bram Adams: **Co-Evolution of Source Code and the Build System**, ICSM 2009: 461-464.

[12] C. Kozyrakis and K. Olukotun: ATLAS: **A Scalable Emulator for Transactional Parallel Systems**, in Workshop on Architecture Research using FPGA Platforms, 11th International Symposium on High-Performance Computer Architecture (HPCA-11 2005), San Francisco, CA, Feb. 13, 2005.

[13] **CDT**: http://www.eclipse.org/cdt/, April 2011.

[14] Celina Gibbs, Jennifer Baldwin, Nieraj Singh, Maja D'Hondt and Yvonne Coady: **Living with the Law: Can Automation Give Us Moore for Less?** ASE '08 Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, 2008.

[15] **CVS**: http://savannah.nongnu.org/projects/cvs, April 2011.

[16] D. Geer: **Eclipse Becomes the Dominant Java IDE**. Computer, Volume 38, Issue 7, 16-18, July 2005.

[17] D. L. Atkins, A. Mockus, and H. P. Siy: **Measuring Technology Effects on Software Change Cost**, Bell Labs Technical Journal, 2000.

[18] Diomidis Spinellis: **Global Analysis and Transformations in Preprocessed Languages**, IEEE Transactions on Software Engineering, Vol. 29, No. 11, November 2003.

[19] E. J. Barry, C. F. Kemerer, and S. A. Slaughter. **On the uniformity of software evolution patterns**, In ICSE ’03: Proc. of the 25th Int. Conference on Software Engineering, pages 106–113,Washington, DC, USA, IEEE Computer Society, 2003.

[20] **Eclipse**: http://www.eclipse.org/, April 2011.

[21] Ernst, M. D., Badros, G. J., and Notkin, D: **An empirical analysis of c preprocessor use**, IEEE Transactions on Software Engineering, vol. 28., Dec 2002.

[22] **Extended Backus-Naur Form**: http://www.comsci.us/languages/metalanguage/ebnf.html, April 2011

[23] G. Goth: **Beware the March of this IDE: Eclipse is Overshadowing Other Tool Technologies**, IEEE Software, Volume 22, Issue 4, July-Aug., 108-111, 2005.

[24] Gail C. Murphy, Mik Kersten, and Leah Findlater: **How Are Java Software Developers Using the Eclipse IDE?** IEEE Software Volume 23 Issue 4, July 2006.

[25] **GCC**: http://gcc.gnu.org/, April 2011.

[26] **GMF**: http://www.eclipse.org/modeling/gmp, April 2011.

[27] I. I. Brudaru and A. Zeller: **What is the long-term impact of changes?** RSSE ’08: Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering, pages 30–32, 2008.

[28] **IBM Rational Application Developer**:
http://www.ibm.com/developerworks/rational/library/08/0926_ackerman-mahate/index.html?S_TACT=105AGX15&S_CMP=EDU, April 2011

[29] **IntelliJ**: http://www.jetbrains.com/idea/, April 2011.

[30] Ivan T. Bowman, Richard C. Holt, Neil V. Brewster: **Linux as a Case Study: Its Extracted Software Architecture**, ICSE '99 Proceedings of the 21st international conference on Software engineering, 1999.

[31] **JFace**: http://www.eclipse.org/resources/?category=Jface, April 2011

[32] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, K. Yelick: **The Landscape of Parallel Computing Research: A View from Berkeley**. No. UCB/EECS-2006-183, December 2006.

[33] **Kaffe** JVM: http://www.kaffe.org/, April 2011.

[34] Kim Sebastian Herzig: **Capturing the Long-Term Impact of Changes**, ICSE '10 Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, 2010.

[35] King Chun Foo, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, Ying Zou, Parminder Flora: **Mining Performance Regression Testing Repositories for Automated Performance Analysis**, 10th International Conference on Quality Software (QSIC), 2010.

[36] László Vidács and Árpád Beszédes: **Opening Up The C/C++ Preprocessor Black Box**, Proceedings of the Eight Symposium on Programming Languages and Software Tools (SPLST), 2003.

[37] **Lookahead**: http://www.engr.mun.ca/~theo/JavaCC-FAQ/kens-javacc-lookahead-summary.txt, April 2011

[38] M. de Jonge: **Build-level components**. IEEE Trans. Softw.Eng., 31(7):588–600, 2005.

[39] M. Haupt, B. Adams, S. Timbermont, C. Gibbs, Y. Coady, and R. Hirschfeld: **Disentangling virtual machine architecture**, IET Software: Special Issue on Domain-specific Aspect Languages, 3(3):201–218, June 2009.

[40] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra: **MPI: The Complete Reference**, MIT Press, 1996.

[41] M.-A. D. Storey K. Wong H. A. Muller: **How Do Program Understanding Tools Affect How Programmers Understand Programs?** WCRE '97 Proceedings of the Fourth Working Conference on Reverse Engineering, 1997.

[42] Martin P. Robillard and Gail C. Murphy: **Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies**, Proceedings of the 24th International Conference on Software Engineering, ICSE 2002.

[43] **NetBeans**: http://netbeans.org/, April 2011.

[44] Nieraj Singh, Celina Gibbs, Yvonne Coady: **C-CLR: A Tool for Navigating Highly Configurable System Software**, ACP4IS '07 Proceedings of the 6th workshop on Aspects, components, and patterns for infrastructure software, 2007.

[45] Nieraj Singh, Graeme Johnson, Yvonne Coady: **CViMe: Viewing Conditionally Compiled C/C++ Sources Through Java Tooling**, OOPSLA '06 Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, 2006.

[46] Omar Alam, Bram Adams and Ahmed E. Hassan: **Measuring the Progress of Projects Using the Time Dependence of Code Changes**, ICSM 2009: 329-338.

[47] **OpenBSD**: http://www.openbsd.org/, April 2011.

[48] OpenMP Architecture Review Board. **OpenMP application program interface**. Technical Report 2.5, OpenMP Architecture Review Board, May 2005. http://www.openmp.org/specs.

[49] Q. Tu and M.W. Godfrey: **An integrated approach for studying architectural evolution**, In IWPC ’02: Proceedings of the 10th International Workshop on Program Comprehension, pages 127–136. IEEE Computer Society, 2002.

[50] Rudolf K. Keller, Reinhard Schauer, Sébastien Robitaille, Patrick Pagé: **Pattern-Based Reverse-Engineering of Design Components**, ICSE '99 Proceedings of the 21st international conference on Software engineering, 1999.

[51] S. E. Sim, C. L. A. Clarke, and R. C. Holt: **Archetypal Source Code Searching: A Survey of Software Developers and Maintainers**, Proceedings of International Workshop on Program Comprehension, pages 180–187, Ischia, Italy, June 1998.

[52] **SOLIDSOURCE SolidSX**: www.solidsourceit.com, April 2011.

[53] Stéphane Ducasse, Tudor Gîrba: **Modeling Software Evolution by Treating History as a First Class Entity**, Workshop on Software Evolution Through Transformation (SETra 2004), pages 71–82, 2004.

[54] Thomas Eisenbarth, Rainer Koschke, Daniel Simon: **Derivation of Feature Component Maps by means of Concept Analysis**, CSMR '01 Proceedings of the Fifth European Conference on Software Maintenance and Reengineering, 2001.

[55] **UML**: http://www.uml.org/, April 2011.

[56] Václav Rajlich, Norman Wilde: **The Role of Concepts in Program Comprehension**, IWPC '02 Proceedings of the 10th International Workshop on Program Comprehension, 2002.

[57] Wuu Yang: **Table-driven look-ahead lexical analysis**, 1994.

[58] **XML**: http://www.w3.org/XML/, April 2011.

Appendix

<table>
<tr>
<td>AND = "&&";
OR = "||";
NOT = "!";
BITWISENOT = "~";
BITWISEAND = "&";
BITWISEOR = "|";
BITWISEXOR = "^";
LEFTPAR = "(";
RIGHTPAR = ")";
GREATERTHAN = ">";
LESSTHAN = "<";
GREATERTHAN_OREQUAL = ">=";
LESSTHAN_OREQUAL = "<=";
EQUALITY = "==";
INEQUALITY = "!=";
POUND = "#";
ADDITION = "+";
SUBTRACTION = "-";
MULTIPLICATION = "*";
DIVISION = "/";</td>
<td>MODULUS = "%";
BITWISE_SHIFT_LEFT = "<<";
BITWISE_SHIFT_RIGHT = ">>";
DOUBLE_QUOTE = "\"";
SINGLE_QUOTE = "'";
DEFINED = "defined";
IF = "if";
IFDEF = "ifdef";
IFNDEF = "ifndef";
ELIF = "elif";
ELSE = "else";
ENDIF = "endif";
PDEFINE = "define";
INCLUDE = "include";
UNDEF = "undef";
PRAGMA = "pragma";
ERROR = "error";
ASSERT = "assert";
UNASSERT = "unassert";
LINE = "line";</td>
<td>IMPORT = "import";
IDENT = "ident";
WARNING = "warning";
SCCS = "sccs";
START_SLASHSTAR = "/*";
END_SLASHSTAR = "*/";
DOUBLE_FORWARD = "//";
LINE_CONTINUATION = "\\";</td>
</tr>
</table>

Table 7: Token types defined for CPP directives.

```
public class LevenshteinAnalyser
    implements ISemanticAnalyser {

    public static int ERROR = -1;

    private int[][] distance;

    private int size;

    public LevenshteinAnalyser(int size) {
        this.size = size + 1;
        distance = new int[this.size][this.size];
    }

    private int minimum(int a, int b, int c) {
        if (a <= b && a <= c)
            return a;
        if (b <= a && b <= c)
            return b;
        return c;
    }

    public int compare(String str1, String str2) {
        if (str1 == null && str2 == null) {
            return 0;
        }
        if (str1 != null && str2 != null) {
            return computeLevenshteinDistance(str1, str2);
        }
        return ERROR;
    }

    private int computeLevenshteinDistance(String str1, String str2) {
        int length1 = str1.length();
        int length2 = str2.length();
        if (length1 > size - 1) {
            length1 = size - 1;
        }
        if (length2 > size - 1) {
            length2 = size - 1;
        }

        for (int i = 0; i <= length1; i++) {
            distance[i][0] = i;
        }

        for (int j = 0; j < length2 + 1; j++) {
            distance[0][j] = j;
        }

        for (int i = 1; i <= length1; i++) {
            for (int j = 1; j <= length2; j++) {
                distance[i][j] = minimum(distance[i - 1][j] + 1,
                    distance[i][j - 1] + 1, distance[i - 1][j - 1]
                        + ((str1.charAt(i - 1) == str2.charAt(j - 1))
                            ? 0 : 1));
            }
        }

        return distance[length1][length2];
    }

}
```

Figure 43: Levenshtein analyser used by the CPP query.