# C-CLR: A Tool for Navigating Highly Configurable System Software

Nieraj Singh, Celina Gibbs, Yvonne Coady
Department of Computer Science
University of Victoria, Canada
{nierajsi, celinag, ycoady}@cs.uvic.ca

## ABSTRACT

In order to accommodate the spectrum of configuration options currently required for competitive system infrastructures, many systems leverage heavy usage of C preprocessor controlled conditional compilation. Inherent costs associated with this heavy preprocessor usage include both the impaired readability of the base system, and the reduced reusability of the configuration code.

Our proposed solution, C-CLR, allows developers to sift through views of a system based on configuration options. Configuration-specific views improve readability of the system as a whole by including only relevant code. They also support reusability by aiding aspect mining through easy navigation to relevant configuration options, and automated identification of equivalent blocks of code within conditionally compiled segments.

### Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.3.3 [**Programming Languages**]: Language Constructs and Features; D2.6 [**Software Engineering**]: Programming Environments.

### General Terms

tools, preprocessor directives, structured programming

### Keywords

system configuration, aspect-oriented programming, modularization, conditional compilation

## 1. INTRODUCTION

Systems code has traditionally achieved clean separation of configuration concerns at the level of the binary – not in the system's source. This is because highly configurable systems tend to leverage C preprocessor (CPP) controlled blocks of code to fine tune conditional compilation. Though practical in terms of performance, modern configurable systems are becoming saturated with preprocessor directives. Conditionally compiled configuration concerns not only impair the readability of the base system, but also reduce the reusability of the configuration code.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Workshop ACP4IS '07, March 12-13, 2007 Vancouver, British Columbia, Canada
Copyright 2007 ACM 1-59593-657-8/07/03... $5.00.

For example, Table 1 overviews the use of conditional compilation in the Harmony [1] open source Java SE project. These measures demonstrate the heavy usage of conditional compilation in this highly configurable JVM. System wide, there are 261 different flags, with 25 of them present in more than 5 of the 682 files in the system. Roughly 60% of the system, 404/682 files, contains at least one conditionally compiled code block.

Unfortunately, a distinguishing characteristic of significant number of configuration concerns is that the degree of tangling with the base code and other configuration concerns, coupled with the critical need for performance, makes attempts to achieve better separation at the level of source very difficult. To help mitigate these difficulties, we propose *C-CLR,* tool-support for sifting through modern configurable system software.

This paper starts by providing an analysis of this configuration problem as it manifests itself in Harmony (Section 2). Based on the results of this analysis, we present the design and implementation of C-CLR (Section 3). C-CLR allows developers to sift through views of the system based on configuration-specific flags, and mine for aspects while navigating the code. An evaluation of using C-CLR with Harmony reveals the challenges of applying aspects in this domain, and how a traditional joinpoint model will not allow us to effectively manage fine-grained interaction of configuration concerns (Section 4).

**Table 1. Conditional Compilation in the Harmony VM, excluding integer controls (e.g, 0 or 1)**

| METRIC | HARMONY |
|---|---|
| Conditional compilation flags | 261 |
| Flags affecting more than 5 files | 25 |
| Flags affecting more than 10 files | 5 |
| Files (.c and .h only) | 682 |
| Files containing CPP conditionals | 404 |

## 2. CONDITIONAL CONFIGURATION

This section introduces the two main problems we have identified with conditional configuration, and provides an

analysis of how these problems manifest themselves in Harmony, an open source JVM written in C.

In terms of configurable systems, it is common to see preprocessor directives throughout the code base. For example, there are 53 instances of conditionally compiled code segments controlled by *#if defined(LINUX)* in Harmony.

We have surveyed and categorized configuration options typically provided by preprocessor directives. They are most commonly used to: (1) redefine macros for alternative configurations, (2) compose multiple configuration options. The following subsections expand on each of these categories, respectively.

## 2.1 Multiple Definitions

As a general design principle there is a trend toward isolating platform-specific behaviour (e.g. opening a file or a socket) behind a well-defined API boundary, as in the Apache Portable Runtime [2], in the Harmony Portability Library. The use of a function-based API enables a higher degree of behavioral selection or replacement of functions at link-time, load-time and runtime. In terms of load-time options, alternative implementations of an API can be loaded using shared libraries (e.g. .dll, .so). Runtime options can be equally flexible, as functions can be called from a table and replaced dynamically. This is the most powerful option and allows injection of memory-overrun checking without relinking the program.

In addition to providing greater flexibility in assembling software components, the API-based model keeps code readable, even at performance critical, low-level parts of the system, where macros are most commonly encountered. Figures 1 and 2 show this in the case of redefining macros for semaphore support for configurations involving Linux and Windows respectively.

```
/* 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 1. Macros in linux/thrdsup.h**

In the Linux build, we see that the macros are defined either as containing functionality if the *LINUX* flag is defined, or empty otherwise (as indicated by *#else* in Figure 1). In the Windows build, we see the macros have a single definition (Figure 2). This means that, in the case of Linux, the developer not only must know which header (.h) files are included in the build, but also how the *LINUX* flag controls the implementation of these macros. It is thus critical for developers to be able to quickly determine the ramification of flag settings in terms of what code is included in a build.

```
/* SEM_CREATE */
/* Arbitrary maximum count */
#define SEM_CREATE(inval)
CreateSemaphore(NULL,inval,2028,NULL)

/* SEM_INIT */
#define SEM_INIT(sm,pshrd,inval)
(sm != NULL) ? 0: -1
```

**Figure 2. Macros in windows/thrdsup.h**

## 2.2 Complex Compositions

Previous work has demonstrated the many ways in which preprocessor directives have proven to be beneficial in implementing conditional code inclusion prior to the compilation phase [5, 11]. When multiple configurations of a system must be supported, conditional compilation is highly beneficial in maintaining a master code base from which multiple versions of a program with different performance, size, or functional characteristics can be constructed. This gives the developer fine-grained control over code placement and access to program state in the form of local variables and arguments. Of course these same characteristics make aspects challenging to introduce into this domain. Further, the intricate composition of configurations poses a significant barrier for aspects.

Preprocessor directives can be used to compose configuration concerns, using traditional logical operators such as *&&*, *//*, and !, as demonstrated in Figure 3. Though there are only a total of five pre-processor flags involved in this code, they demonstrate the intricate composition often required for effective configuration strategies. In this example the outer nested condition, *LINUXPPC // PPC*, groups these sub-configurations together conceptually, into what can be considered as a higher-level configuration. Within this nested conditional, this higher-level configuration is then split along yet another distinct set of sub-configuration lines, *IBMC // IBMCPP* verses *LINUX*.

```
#if defined(LINUXPPC) || defined(PPC)
  int ppcCacheLineSize;
  int i;
  int input1 = 20;
  char buf[1024];
  memset (buf, 255, 1024);
#if (__IBMC__||__IBMCPP__)
  dcbz ((void *) &buf[512]);
#elif defined(LINUX)
__asm__ ("dcbz 0, %0":         /* no outputs */
:"r" ((void *) &buf[512]));
#endif
for (i = 0, ppcCacheLineSize = 0; i < 1024; i++)
    {
      if (buf[i] == 0)
        {
          ppcCacheLineSize++;
        }
    }
  PPG_mem_ppcCacheLineSize = ppcCacheLineSize;
#endif
  return 0;
}
```

**Figure 3. Composed and nested conditionals in hycpu.c**

# 3. C-CLR

In an effort to aid developers working with configurable systems with a heavy preprocessor presence in terms of conditional compilation, we have developed *C-CLR* – a tool for navigating sources based on configuration flags and their

values. Our goal in the development of C-CLR was twofold. First we wanted to make it easier for developers to view only code segments relevant to a particular flag setting, and second to support the mining of aspects in these configurable code bases.

C-CLR is an Eclipse plugin written in Java and consists of two main components: (1) a lower level lexical analyzer and parser, and (2) higher level semantic object model of the source. The latter includes a C preprocessor directive symbol table and a separate AST indexer.

## 3.1 Lexical Analyzer and Parser

The parsing algorithms have been optimized for C preprocessor parsing and occur in a streamed, one-pass process, using standard compiler front-end principles. The result of the lower level is a compacted AST where a source is modeled in terms of two entities: common code blocks, and those that are conditionally compiled. The block nodes themselves are offset and length values compatible with the *org.eclipse.jface.text.IDocument* and *org.eclipse.jface.text.IDocumentPartitioner* interfaces, and the Eclipse editor framework in general. String and object instantiations are kept at a minimum so that the indexer containing the ASTs for all the resources in a C project result in low, long-term heap allocations.

## 3.2 Semantic Object Model

C-CLR translates C sources into semantic object models where traversal of the latter generates a particular view of a source based on a set of defined flags. These sources can then be mined for common patterns controlled by C preprocessor directives. Our intuition for this feature is that, since configuration in C-based JVMs like Harmony is largely implemented via conditional compilation, it may be of general interest to developers to analyze whether conditionally compiled blocks share a common structure. Given that this code is traditionally scattered and tangled, it may also be of interest to establish the applicability of aspects within configuration-dependant code, and help clarify design and implementation issues.

## 3.3 Sifting with C-CLR

C-CLR parses from a source and displays all C preprocessor flags that control conditionally compiled code blocks. Figure 4 shows a C-CLR view where a subset of flags parsed from the Harmony source can be selected and evaluated by the C-CLR semantic model layer.

It is important to note that, although semantically correct C views are generated with any fine-grained permutation of C preprocessor flags, C-CLR does not currently indicate flag dependencies and thus views generated by the tool may not translate to actual functional builds.

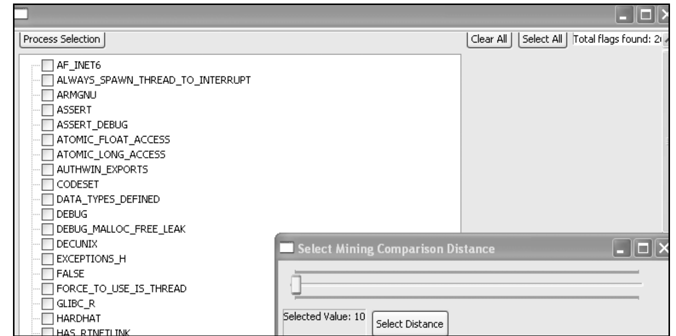


**Figure 4. Flag Selection in C-CLR.**

As an alternative to the default flag view, an external configuration header file (typically a .h) used in the actual build process can be read by C-CLR, and valid flag options grouped and displayed in specification categories. Rudimentary flag dependencies can also be grouped in an XML file validated by a specific schema and parsed by C-CLR.

The mining process can be configured by specifying the syntactic distance between a potential match and a block pattern. This distance is similar to a Levenshtein distance [7], but it factors out white spaces and comments. Figure 4 shows where this distance can be specified. Although not shown in the screenshot, upper and lower bounds on pattern redundancies can also be set through the tool. For example matches can be found for conditionally compiled blocks of code that are present more than 10 times but less than 15.

Figure 5 shows the result of mining the flag configuration specified in Figure 4. Here two different views of the *hysignal.c* file from the Harmony source are displayed, the left showing the flags chosen for the configuration displayed at the top of the view, and the right highlighting matched common patterns. The results tree, shown in the top section of Figure 5 displays mining matches based on common patterns found by C-CLR. Navigation to the exact locations of the matches in the source view is possible directly from the results tree.

In the example shown, the pattern:

**name="RDI"; *value=&info->sigContent->rdi;*

was chosen from the tree, and all matches in the selected file highlighted. Note that pattern matches are not exact since a comparison distance greater than zero was selected in Figure 4. This allows more flexibility in finding patterns that may be structurally equivalent but vary in variable names or order. Simply sliding the distance to zero can attain exact pattern matches.

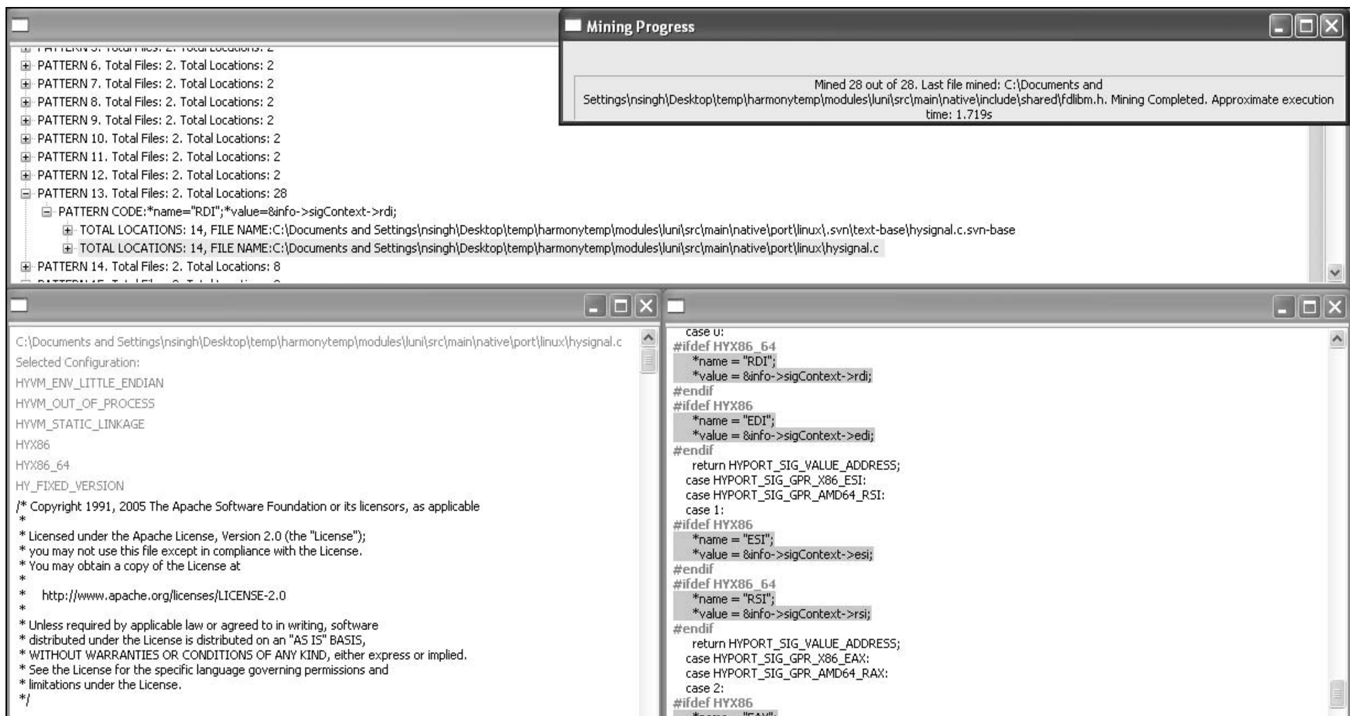


**Figure 5. Mining results of flag selection from Figure 4**

# 4. EVALUATION

This section evaluates two open source virtual machine projects, Harmony [1] and the K Virtual Machine (KVM) [13]. Harmony is a Java Platform Standard Edition project of the Apache Software Foundation, which identifies the development of a modular runtime architecture as a key system goal. KVM is Sun's open source Java 2 Platform, Micro Edition (J2ME) project with a portable design that minimizes system dependencies while adhering to strict constraints on footprint size. We consider both these systems to be representative of modern, configurable system infrastructures.

We now consider each of the configuration categories identified in Section 2, and evaluate the impact of C-CLR in each case.

## 4.1 Distinctive Definitions

As previously established in Section 2.1, Harmony uses macros to achieve configurability that is essentially API-based for some low level features such as system semaphores. In Harmony, there are many examples where macros are conditionally redefined for specific sets of sub-configurations in one file. That is, a single module contains multiple sets of conditionally compiled macro definitions. In this case where all sub-configurations include the header file associated with Linux, but the *LINUX* flag may or may not be set.

```
/* SEM_CREATE */
#if defined(LINUX)
#define SEM_CREATE(initValue)
        thread_malloc(NULL, sizeof(OSSEMAPHORE))

/* SEM_INIT */
#if defined(LINUX)
#define SEM_INIT(sm, pshrd, inval)
        (sem_init((sem_t*)sm, pshrd, inval))
```

**Figure 6. C-CLR view of linux/thrdsup.h, LINUX flag set.**

C-CLR supports developers working with sub-configurations introduced by redefinition of macros, narrowing a developer's view to a single sub-configuration, dependant on the flags selected. Figure 6 illustrates this narrowed view of semaphore macro definitions for configurations that include *linux/thrdsup.h* when the *LINUX* flag is set. As shown in Figures 7 and 8, C-CLR views can be toggled to hide/display code that is irrelevant to a given flag setting. Thus, though the source specifies multiple definitions for this macro, the C-CLR view shows only the one.

## 4.2 Clarification of Configurations

Figure 7 shows the use of nested and composed conditionals that illustrate the complexities associated with dependencies between flags. The C-CLR view will identify the matching conditional expressions. In this example, the *LINUX* and *LINUXPPC*, but not the *IBMC* or *IBMCPP* flags are set. This begins to show how some flags (*LINUX*, *IBMC*, *IBMCPP*) are dependant on a combination of other flags (*LINUX*, *LINUXPPC*).

**Figure 7. Composed conditionals within Harmony.**

Figure 10 also illustrates the dependency between the *LINUX* and *HAS_RTNETLINK* flags and highlights the need for a more coarse-grained flag selection, which takes these dependencies into account. In this example the *HAS_RTNETLINK* flag was not selected in C-CLR's flag selection process, while the *LINUX* flag was. C-CLR therefore shows all code wrapped in the *HAS_RTNETLINK* conditional in gray font. If you look closely though, the first conditional code introduced by the *LINUX* flag in fact set the *HAS_RTNETLINK* flag, yet the *HAS_RTNETLINK* conditional code still remains gray.

## 4.3 Analysis

In each of the three problems associated with conditional configuration C-CLR has demonstrated improved support for highly configurable systems.

With respect to redefined macros, developer's can use C-CLR to quickly identify which definition is associated with a given flag, but not all definitions are associated with flags. Relevant header files are also difficult to determine. In the example of the Windows definitions in Figure 2, no conditional compilation is used in the header file, and thus C-CLR does not navigate these definitions.

With respect to the one to many relationship between a single configuration flag and concerns that could be structured as aspects, C-CLR automatically seeks out potential crosscutting structure as redundant code blocks, and further provides effective navigation for manual inspection of conditional compilation according to where a given flag applies. So in the case where a flag covers a large quantity of unrelated code segments, such as the 53 instances of *#if defined(LINUX),* C-CLR assists developers in mining the candidates, so that they can more easily be considered for refactoring as aspects.

Finally, with respect to what appears to be the most challenging problem – the complex composition of configurations – C-CLR offers the ability to iteratively refine navigation expeditions in order to improve reasoning about interactions experimentally through flag assignments. In its current form however, C-CLR does not supply any additional semantic leverage to understand dependencies between flags or the configurations to which they belong.

**Figure 8. Hidden vs. full view for the same file.**

# 5. RELATED WORK

The problem of heavy preprocessor presence in source has been tackled via a number of different tools. In modern development environments, preprocessor tooling may be coupled with the IDE being used for development. Tool support for preprocessors that perform source-to-source transformations include Cppp, sunifdef [20], Emacs' hide-ifdef-mode, and CTAGS/GTAGS [10, 6]. László Vidács and Árpád Beszédes presented a general schema solution called CANPP that generates a tree from the object models of a source's tokens [22]. Particular views of a master source can be extracted via CANPP and directive metadata appear to be recoverable from the latter.

Our approach is different to all of these in the sense that we have designed our conditional compilation development environment to be used with highly portable Java-based IDE, Eclipse, and provide a medium that takes advantage of the rich refactoring and editing tools available in modern IDEs. In particular, with respect to aspects, we envision support for ACDT much like AJDT. The motivation behind this decision was in part to specifically target modern development teams familiar with the benefits of tool support for systems development.

In terms of the application of aspects within this domain, we anticipate the granularity of the join points for many concerns will be a problem, as they cannot often be defined in terms of method signatures. In [18], Siadat *et al* provide results that suggest an intolerable amount of refactoring to expose sufficient joinpoints in systems code. In [4], Colyer *et al* applied AOSD at the level of middleware systems both at a coarse-granularity of feature integration and a fine-granularity of product line policy application. This case study illustrated the tradeoffs associated with using aspects to localize heterogeneous concerns – those which exhibit diverse behaviour at each interaction point. We believe effective refactoring techniques will be necessary to avoid both the tedious process and lack of benefits associated with large, unwieldy aspects with a 1:1 pointcut:advice ratio. We believe

that these results, coupled with our own experience in this domain with the Jikes RVM [12], suggest a need for a finer-grained join point model, while still providing a higher-level abstraction for points of execution in systems infrastructure.

## 6. CONCLUSIONS

The problems associated with conditionally compiled configurations include redefined macros, non-reusable configuration code, and non-explicit representation of configuration compositions. C-CLR fits within reasonable parameters of performance for a build process, and can allow developers to navigate, mine for aspects, and explore complex compositions associated with conditional compilation. C-CLR does not, however, supply any extra semantic leverage to understand dependencies between flags or the configurations to which they belong. We believe this, coupled with the need for a modified join point model within this domain, to be the most significant future challenges associated with this work.

## 7. REFERENCES


[1] Apache Harmony, http://incubator.apache.org/harmony/.

[2] Apache Portable Runtime Project, http://apr.apache.org/.

[3] Yvonne Coady, Gregor Kiczales, Mike Feeley and Greg Smolyn, Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code, Proceedings of the Joint European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9), 2001.

[4] Adrian Colyer and Andrew Clement, Large-scale AOSD for Middleware, Proceedings of ACM International Conference on Aspect-Oriented Software Development (AOSD), 2004.

[5] Michael D. Ernst, Greg J. Badros, and David Notkin, An Empirical Analysis of C Preprocessor Use, IEEE Transactions on Software Engineering, Vol. 28, No. 12, December 2002.

[6] Exuberant Ctags, http://ctags.sourceforge.net/.

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

[8] Celina Gibbs, Chunjian Liu and Yvonne Coady, Sustainable System Infrastructure and Big Bang Evolution: Can Aspects Keep Pace? European Conference on Object-Oriented Programming (ECOOP), 2005.

[9] Celina Gibbs, Yvonne Coady, Michael Haupt, Jan Vitek and Hiroshi Yamauchi, A Domain Specific Aspect Language for Virtual Machines, in DSAL 2006.

[10] GNU Global Source Code Tag System, http://www.gnu.org/software/global/.

[11] Ying Hu, Ettore Merlo, Michel Dagenais, and Bruno Lag¨ue. C/C++ Conditional Compilation Analysis Using Symbolic Execution, Proceedings of the international Conference on Software Maintenance (ICSM), 2000.

[12] IBM, Jikes Research Virtual Machine, www-124.ibm.com/developerworks/oss/jikesrvm/, 2004.

[13] Sun Microsystems, The K Virtual Machine, http://java.sun.com/products/cldc/ds/.

[14] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswold, An overview of AspectJ, Proceedings of 15th European Conference on Object-Oriented Programming (ECOOP), 2001.

[15] M. Krone and G. Snelting, On the inference of configuration structures from source code, Technical Report 93-06, Gaussstrasse 17, D-38092 Braunschweig, Deutschland, 1994.

[16] David Larochelle and David Evans, The Splint Manual, Version 3.1.1 27 April 2003, http://lclint.cs.virginia.edu/manual/manual.html#operation

[17] Mario Latendresse, Fast Symbolic Evaluation of C/C++ Preprocessing Using Conditional Values, Proceedings of the Seventh European Conference on Software Maintenance and Reengineering (CSMR), 2003.

[18] Jamal Siadat, Robert Walker, Cameron Kiddle. Optimization Aspects in Network Simulation. Proceedings of ACM International Conference on Aspect-Oriented Software Development (AOSD), 2006.

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

[20] Sunifdef, http://www.sunifdef.strudl.org/.

[21] Peri Tarr and Harold Ossher, Hyper/J User and Installation Manual, www.research.ibm.com/hyperspace, 2000.

[22] 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.