A Taxonomy of Comments

Or a Critique of C and OS Code Through Comments

Yoann Padiolean and Lin Tan

September 10, 2008

Contents

Chapter 1

Overview

There are different ways to classify comments, depending on the questions we are interested in:

- **What?** What is inside the comment ? Why it was written ? Does it contain useful information ? If yes who is the target of the comment, the programmer working with (client) or working on the code (implementer), or possibly a tool ?
- **Where?** Where is the comment? In our case in which OS (Linux, FreeBSD, OpenSolaris), in which subsystem (core, driver, filesystem, network protocol, etc), in an implementation (.c) or header (.h) file, and its place in the file (in a header, function, structure, macro, etc).
- **When?** When the comment was written? It can be the absolute time (10 years ago), or the time relative to the file creation (2 months after file creation). The last information can give a hint about the development phase (design, maintenance, etc) the comment was written for, the first if those kinds of comment are still relevant today.
- **Who?** Who is the author of the comment ? A core developer? A maintainer? A tester? A beginner? An expert? Or maybe it was auto-generated by a tool ?

From those four general and comprehensive questions, the last three can be to some extent automatically answered; they are mainly quantitative. Some previous works partially studied those three questions, for instance [] was interested in the difference between expert and beginners comments. Section **??** discusses those related works. We present the results of our own study where we analyze the source code repositories of the three OS in Section **??**.

In this section we are interested in the first question, the **what**, arguably the most important one considering our main goal which is to listen to programmers. This question is mainly qualitative and as such more difficult, which may explain why nobody before to the best of our knowledge have really studied in depth this question. For this we have manually examined 1050 comments randomly sampled from the three OS and classified them in a taxonomy we gradually refined, as explained in Section **??**.

The toplevel categories of our taxonomy are:

- *Meta Information*
- *Past and Future*
- *Explanation*
- *Type*
- *Interface*
- *Code Relationships*

- *Other*

The following sections will detail each of those toplevel categories. The most interesting one for bug-checking are *Type*, *Interface*, and *Code Relationships*.

One way to view those toplevel categories is that they roughly correspond to the order in which one can see them in a file. The header of the file usually contain a copyright notice and other *Meta Information*, then come some log about the history of the file, its *Past and Future*, then a summary or high-level *Explanation*, then come some data-structures and *Type* definitions, and then some functions and their *Interface*. The primary purpose of comments is to help understand code, and as such the last two categories are used to help the programmer understand an entity in isolation, without even looking at the implementation. But at some point, to understand a program, one has to understand how entity works together and so need to understand *Code Relationships*. In some way *Past and Future* can also be seen as describing relationships, how the current code relates to its past and future.

This is of course only a rough correspondence. In practice some notions crosscut; a copyright can for instance also be attached to a specific function. As opposed to the 'where', 'when', and 'who', which lead to obvious classifications, the 'what' is more difficult to classify as comments are used for a wide range of purpose.

Also, some operating system notions *crosscut* those categories. We repeatedly encountered mainly four topics in OS comments:

- Resource management: especially about memory and buffer handling
- Timing: concurrency, complex flow interaction, lock, interrupt
- Low-level interaction: bit and byte layout, hardware register interaction, network format, endianess
- Protection: address space, scope

For instance concurrency comments are sometimes used in *Interface*, to describe the assumptions made by a function about locking, but also to describe *Code relationship*, such as how a lock variable is related to and protect other variables.

Note that some comments may be obsolete or even misleading, but for our purpose, which is to identify programmer needs, this is not an issue. Indeed those comments were certainly at one point correct and so expressed a valid programmer need.

In the same way even if we only analyze the comments present in the current version of the three OS, and so may miss some comments that have disappeared, we think that they were probably not important comments. They may have disappeared because they expressed a past programmer need that was fulfilled by a programming language extension or tool.

By looking at those comments, we found that many of them could or should be supported by better or existing:

- Programming languages (PL), especially:
  - better type system
  - programming features
- Software development tools, especially:
  - Bug detection tools (checkers)
  - Source code visualization
  - Source code navigation
  - Version control system (VCS)
  - Bug database

  - Collaboration tools
  - Debugger and Profiler
  - Tester (regression testing framework)
  - Integrated Development Environment (IDE) which encompasses most of the preceding items
- Software process methodologies

The line between a bug detection tool (checker), a (type-based) annotation language, and a programming language type system is a very thin line. What is put inside the compiler and type system and what is put outside of it in external tools may change. Also, even programming language features can now be put in external tools like Xoc [1]. That's why annotation languages are not part of the preceding list. Also, "annotation" is a too general term. Annotations are not used only for bug finding. Annotations crosscut the preceding list. Multiple tools already use special annotations in comments:

- Documentation tools use special annotations for authorship
- CVS use special annotations (`$Id:$`, `$Log:$`)
- Editors like Eclipse or Emacs use special annotations to better visualize the source code or manage TODO items
- Some research work on debugging (KStruct []) proposed to add annotations on fields in structure to help debugging, and MicroDriver [] proposed annotations to specify what part of the driver must be put in kernel space and what part can be in user-space

This in fact shows that special annotations in comments are already used for a wide range of purpose, and we think may be generalized to more uses. Comment annotations can be used as a special artifact in the code for different tools to collaborate with each other: comments can be *the ring that rule them all*. For instance the bugzilla tool could extract automatically, thanks to a special comment annotation, the maintainer name of a file to automatically send the bug report to the right person.

```
/*
 * This comment is parsed by configure to create ctype.c,
 * so don't change it unless you know what you are doing.
 *
 * .configure. strxfrm_multiply_cp932=1
 * .configure. mbmaxlen_cp932=2
 */
```

```
if (ip_sock == INVALID_SOCKET)
    {
      DBUG_PRINT("error",("Got error: %d from socket()",socket_errno));
      sql_perror(ER(ER_IPSOCK_ERROR));              /* purecov: tested */
      unireg_abort(1);                              /* purecov: tested */
    }
```

So, in the rest of the rest paper, the terms comments, special comments, special annotations, or annotations are considered equivalent.

Remember that comments are mostly used as an escape door by the programmer, because there is no other way (no PL feature, no tool) to express what the programmer has in his mind.

Chapter 2

Meta Information()

The *Meta* category allows the programmer to express his need to identify and protect his work. Programming languages (PLs) do not allow to express such needs, except maybe Eiffel [].

There are mainly 3 subcategories but they are mostly used at the same time in the same comment:

- *Author*
- *Copyright*
- *Date*

```
/* $Id: mntfunc.c,v 1.19.6.4 2005/01/31 12:22:20 armin Exp $
 *
 * Driver for Eicon DIVA Server ISDN cards.
 * Maint module
 *
 * Copyright 2000-2003 by Armin Schindler (mac@melware.de)
 * Copyright 2000-2003 Cytronics & Melware (info@melware.de)
 *
 * This software may be used and distributed according to the terms
 * of the GNU General Public License, incorporated herein by reference.
 */
```

```
/* ...
 * Send bug reports and improvements to <boggs@boggs.palo-alto.ca.us>.
 */
```

Those kinds of comments are now more formally supported by tool like Javadoc (`@author`, `@copyright`, etc), which confirms that the repeated use of special comments in the past led to the invention of special annotations and tools.

Also, tools like CVS can automatically adjust some comments containing special tags (`$Id:$`, `$Log:$`) to display for instance the last person who modified the file, which shows that comments can be used as the basis for tool cooperation.

Some copyright notice are sometimes enclosed by special tags, like in OpenSolaris (`/* CDDL HEADER END */`), which can be used for instance by tool like Emacs to automatically hide the copyright if the programmer want to focus on the code.

```
!!/*
   * CDDL HEADER START
```

```
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License, Version 1.0 only
 * (the "License"). You may not use this file except in compliance
 * with the License.
 ...
 * CDDL HEADER END
 */
/*
 * Copyright 2002 Sun Microsystems, Inc. All rights reserved.
 * Use is subject to license terms.
 */
```

But those meta-information are right now mainly used for doc-generation. One could imagine that if a bug-database tool like bugzilla knew about those meta-information, then a bug report could be automatically forwarded to the right person, providing the comment also contains email information, if not of the author maybe of the maintainer of the file. If one would like to do a survey and ask questions to kernel programmers, those annotations may also be useful to automate the process.

Also, even if some file may be owned by multiple authors, with different copyright, there is no easy way to actually know which parts belong to which authors by just inspecting the current version. Some meta-information can and are sometimes associated to specific functions to identify fine-grained ownership, but it is rarely used as it is tedious to repeat to dozen of functions.

```
/*
 * David H rdeman <david@2gen.com>
 * The key makes the SCSI stack print confusing (but harmless) messages
 */
UNUSUAL_DEV( 0x4146, 0xba01, 0x0100, 0x0100,
             "Iomega",
             "Micro Mini 1GB",
             US_SC_DEVICE, US_PR_DEVICE, NULL, US_FL_NOT_LOCKABLE ),
```

There is no easy way to describe the *scope* of the code owned by different authors. As such it can also be difficult for tools to identify the copyright of a specific part of a multi-authors code, which depending on the license, for instance GPL or public domain, may have a strong impact. Some companies for instance want to detect if the code of their employees are actually copy-paste of codes with restrictive (like GPL) license.

In fact more recently advanced VCS allow for each line of a file to easily know who has written this line and when (with `cvs annotate foo.c`) which can be very useful when debugging code with lots of authors to know who is responsible for a buggy code and so who should be contacted to fix it. But the algorithm used is line oriented and may be messed by simple cut and paste. It is actually difficult to track the author of a line by just using the information in a VCS [**?**]. Maybe some support from the IDE may help, where the authorship will be more explicit, but hidden from the programmer that would not like to be bothered by such authorship marks. But with those marks in the text, it would be trivial to track the author through cut and paste.

The emacs library used a special format of comments which can be understood by some tools to help programmers find plugins.

```
;;; dircolors.el -- provide the same facility of ls --color inside emacs
;; Emacs Lisp Archive Entry
;; Filename: dircolors.el
;; Author: Padioleau Yoann <padiolea@irisa.fr>
;; Version: 1.0
```

Microsoft [] have proposed also to add *manifest* to code as self-describing artifact, specifying what the code is for.

```
/*
    DO NOT EDIT THIS DOCUMENT !!! THIS DOCUMENT IS GENERATED BY
    mozilla/intl/unicharutil/util/genbidicattable.pl
 */
```

Chapter 3

Past and Future()

The *Past and Futur* category allows the programmer to express his need to refresh his own memory. Robery Warren [] said: "comments are both a memory aid to the original developers and a guide to future source code readers". It can also be a guide for futur changes.

Programmer use comments for such purpose as programming language offer few features to talk about the past and future of the code. An exception may be the Eiffel programming language which allows to annotate functions as deprecated.

Even if people can now use version control system (VCS) to go back in time, programmer still prefer or at least still use comments for talking about the past, be it to show old code statements, changelogs (which could be extracted from commit messages), or notes about the past.

3.1 Todo()

Programmers use different forms of TODO (/* TODO */, /* FIXME */, /* XXX */).

```
    buf->vb.field  = field; // FIXME: check this
```

```
/*FIXME: using G rates.*/
```

```
    addr = addr >> 2; /* temporary hack. */
```

```
!!/*
    * Update parameters of an IPv6 interface address.
    * If necessary, a new entry is created and linked into address chains.
    * This function is separated from in6_control().
    * XXX: should this be performed under splnet()?
    */
   int
   in6_update_ifa(struct ifnet *ifp, struct in6_aliasreq *ifra,
```

```
struct ... {
        ...
        /* XXX - most fields in ki_rusage_ch are not (yet) filled in */
        struct  rusage ki_rusage_ch;     /* rusage of children processes */
        struct  pcb *ki_pcb;             /* kernel virtual addr of pcb */
}
```

The preceding comment also expressed a data-flow condition.

```
rpn->param_mask = htole16(param_mask); /* XXX */
```

```
        /*
         * XXX - would be nice if we could do this without suspending...
         */
        txg_suspend(dp);
```

```
!!      /* Original seq number I used ??questionable to keep?? */
        uint32_t init_seq_number;
```

Some of those comments even have authors name attached to and so are used as a communication medium for collaborative work.

```
        /* FIXME: Source route IP option packets --RR */
        if (nf_conntrack_checksum && hooknum == NF_INET_PRE_ROUTING &&
```

Tools like Eclipse now enable to gather many of those special comments in a special "TODO view" with cross-reference capability, a small improvment over manual grepping.

There is no connexion right now between TODO, bugzilla, and VCS. So, there is no way to check that a TODO erased in the past was erased because of a bugfix and that it was legitimate to erase it. It would require of course to understand the TODO, but maybe TODO could have a better format precising more the kind of TODO or priority. For instance We could have a refactoring TODO annotation that could be automatically removed when the refactoring action is performed under the IDE.

Some TODO are also present as macro and used in the concrete code, generating a warning at run-time about the lack of a feature.

linux/drivers/net/wireless/bcm43xx/bcm43xx_debugfs.h

```
#define TODO() \
do { \
printk(KERN_INFO PFX "TODO: Incomplete code \
 in %s() at %s:%d\n", \
__FUNCTION__, __FILE__, __LINE__); \
} while (0)


#define FIXME() \
do { \
printk(KERN_INFO PFX "FIXME: Possibly \
 broken code in %s() at %s:%d\n", \
__FUNCTION__, __FILE__, __LINE__); \
} while (0)
```

Some tools like log4j [] for Java provide more advanced functionalities.

3.2 Reminder()

```
!!      /*
         * We kmem_alloc() the sigaction array because
         * it is so big it might blow the kernel stack.
```

```
         */
        sap = kmem_alloc((NSIG-1) * sizeof (struct sigaction), KM_SLEEP);
```

Some of the bugfix reminder comments could be connected to a bug database to clearly show what bug they fixed.

Some comments are also used to describe the absence of something because as something is absent, there is no other way than using comments to talk about what is missing.

```
!!/*
   * We used to declare this array with size but gcc 3.3 and older are not able
   * to find that this expression is a constant, so the size is dropped.
   */
  extern pgd_t swapper_pg_dir[];
```

Sometimes people also use comments to specify that there is nothing, to represent the emptiness, like in Yacc for some empty rules, to make it clear that it's not an error that the rule is empty but it was done on purpose:

```
NOT FROM SAMPLE (parser_c.mly)
struct_decl_list_gcc:
 | struct_decl_list  { $1 }
 | /* empty */       { [] } /* gccext: allow empty struct */
```

```
  while (...;...;...)
      /* nothing */;
```

```
} else if ((codec72==0x8000) && (codec6c==0x0080)) {
/* nothing */
}
```

```
   struct kmem_list3 *nodelists[MAX_NUMNODES];
        /*
         * Do not add fields after nodelists[]
         */
         /*
         * We put nodelists[] at the end of kmem_cache, because we want to size
         * this array to nr_node_ids slots instead of MAX_NUMNODES
         * (see kmem_cache_init())
         * We still use [MAX_NUMNODES] and not [1] or [0] because cache_cache
         * is statically defined, so we reserve the max number of nodes.
         */
```

```
  /*
   * This code is intentionally commented.  The window proc
   * for the list box implements WM_SETREDRAW to invalidate
   * and erase the widget.  This is undocumented behavior.
   * The commented code below shows what is actually happening
   * and reminds us that we are relying on this undocumented
   * behavior.
   */
//              int flags = OS.RDW_ERASE | OS.RDW_FRAME | OS.RDW_INVALIDATE;
//              OS.RedrawWindow (handle, null, 0, flags);
```

3.3 Trigger, deprecated and obsolete()

Deprecated and triggers are both about software evolution and the two sides of the same problem. One is used to indicate that some code should not be used anymore, and the other to indicate that some code should be added if an "event" happens. This "event" can be a complex condition.

```
my_bool unused0; /* Please remove with the next incompatible ABI change. */
```

```
/*
 *  Scripts for SYMBIOS-Processor
 *
 *  We have to know the offsets of all labels before we reach
 *  them (for forward jumps). Therefore we declare a struct
 *  here. If you make changes inside the script,
 *
 *  DONT FORGET TO CHANGE THE LENGTHS HERE!
 */
```

```
!!      /* This is deprecated, FIOGETOWN should be used instead. */
        case TIOCGPGRP:
```

```
!!#define SAL_ERR_FEAT_LOG_SBES                   0x2     // obsolete
```

Note that gcc allows to annotate some functions as 'deprecated'. People also use cpp ifdef tricks to achieve the same result. Java also allows this, as API evolution is a very important problem. Thanks to the annotation, a warning message is displayed at compilation time to warn the programmer to evolve his code. But there is no support to mark cpp macro as deprecated, as in the preceding comment. This shows again that many comments are about cpp and that there is no tool working at cpp level.

```
!!/*
   * These are the binary operators that are supported by the expression
   * evaluator. Note that if support for division is added then we also
   * need short-circuiting booleans because of divide-by-zero.
   */
  static int op_lt(int a, int b) { return (a < b); }
```

```
!!/* WARNING: If you change any of these defines, make sure to change the
   * defines in the X server file (radeon_sarea.h)
   */
  #ifndef __RADEON_SAREA_DEFINES__
  #define __RADEON_SAREA_DEFINES__
```

```
// plane_t structure
// !!! if this is changed, it must be changed in q_shared.h too !!!
#define pl_normal       0
#define pl_dist         12
```

```
/* Shouldn't this be in a header file somewhere? */
#define BYTES_PER_WORD          sizeof(void *)
```

In this case, with a special annotation, a tool connected to a VCS could regularly check if the same functionality could be covered by an existing more general macro (instead of duplicating the same functionality in many files), and warn the user if this situation happens. In this case the tool could search for a macro with a similar definition by clone detection. The use of the VCS could allow to do this clone detection only on new code, which is an information only the VCS has, which would avoid to rerun expensive analysis on the whole source tree. It's yet another example of possible synergy, here between a clone detection tool and a version control system.

Programmers could also use a tag which is the opposite of deprecated: 'new'. If someone introduces a new functionality, there is often a very long time before the other programmers know about this functionality and refactor old code to make use of this better function or macro. This tag could be used by a tool to make the computer more pro-active. If new code is added and recognized as similar to a functionality provided in a library function tagged with a new, then the IDE could warn the programmer to remove his code and instead use the library function. The comment in the library could help to specify what kind of code should be re-written:

```
/* @evo: n & (n-1) ====> IS_POWER_OF_2(n) */
#define IS_POWER_OF_2 (n) (n) & ((n) -1)
```

The pattern could be written as a SmPL program (cf section **??**).

3.4 OldCode()

Even if many people can now use advanced version control system (VCS), allowing to go back in time, many programmers still prefer to keep some old code in the file in comment. But this clutter code. If there is too many such comments, then it is visually more tedious to understand the code as one is visually bothered by the old code.

Maybe it shows a problem of VCS. The steps to see the old version of some code is maybe too long for the programmer, who prefers to keep at sight this code in comment. Maybe also when he remembers some old code that could be used back, the VCS does not provide any help to find this old code. Also not old code are interesting. Maybe a source code visualizer with the help of the VCS could show old interesting code under the recent code (with some transparency effects one could see both at the same time).

```
        /* force reset on */
        val |= INFINIPATH_SERDC0_RESET_PLL
!!              /* | INFINIPATH_SERDC0_RESET_MASK */
                ;
```

```
                i++;
!!              /* DELAY(100); */
                s = STATUS(m);
```

Note that people may ask what is the unit of DELAY, 100 seconds or milliseconds or nanoseconds ? See the *Unit* category later.

Those old code, sometimes storing debugging instructions, are unfortunately not compiled and as the code evolves, the compiler can not detect legitimate potential errors in such code. In fact, such old commented code may stay in the code for years, and the reason it was put in comment may gradually be forgotten by the original developer or maintainer. Instead of using comments programmers can also used advanced macro like DEBUG or ifdef that can be easily enabled or disabled to generate or not debugging information at run-time. Note also, again because the use of cpp, that gcc has not the opportunity to check such code when the option is disabled.

We didn't find that much *OldCode* comments, which is a little surprising. It would also be interesting to know the age of such comments, as well as the age of TODO comments and their evolution.

3.5 Log()

Programmers add changelog information in comments, for collaboration, for summarizing the set of features added on top of each other.

```
  /* ...
   * ChangeLog
   * Jun 11 2001        Takashi Iwai <tiwai@suse.de>
   *      - Recoded & debugged
   *      - Added timer interrupt for midi outputs
   *      - hwports is between 1 and 8, which specifies the number of hardware ports.
   *        The three global ports, computer, adat and broadcast ports, are created
   *        always after h/w and remote ports.
   *
   */

  #include <linux/init.h>
  #include <linux/interrupt.h>
```

```
!!/*
   * Ok, demand-loading was easy, shared pages a little bit tricker. Shared
   * pages started 02.12.91, seems to work. - Linus.
   *
   * Tested sharing by executing about 30 /bin/sh: under the old kernel it
   * would have taken more than the 6M I have free, but it worked well as
   * far as I could see.
   *
   * Also corrected some "invalidate()"s - I wasn't doing enough of them.
   */
/*
 * Real VM (paging to/from disk) started 18.12.91. Much more work and
 * thought has to go into this. Oh, well..
 * 19.12.91  -  works, somewhat. Sometimes I get faults, don't know why.
 *              Found it. Everything seems to work now.
 * 20.12.91  -  Ok, making the swap-device changeable like the root.
 */

/*
 * 05.04.94  -  Multi-page memory management added for v1.1.
 *              Idea by Alex Bligh (alex@cconcepts.co.uk)
 *
 * 16.07.99  -  Support of BIGMEM added by Gerhard Wichert, Siemens AG
 *              (Gerhard.Wichert@pdb.siemens.de)
 *
 * Aug/Sep 2004 Changed to four level page tables (Andi Kleen)
 */
```

```
/*
 *  linux/init/main.c
 *
 *  Copyright (C) 1991, 1992  Linus Torvalds
 *
```

```
 *  GK 2/5/95  -  Changed to support mounting root fs via NFS
 *  Added initrd & change_root: Werner Almesberger & Hans Lermen, Feb '96
 *  Moan early if gcc is old, avoiding bogus kernels - Paul Gortmaker, May '96
 *  Simplified starting of init:  Michael A. Griffith <grif@acm.org>
 */
```

Unfortunately one can not click on those different log entries to see to what code they correspond too. There is no formal connexion to the VCS. One would like to add some formal log comments that when clicked show the corresponding modifications in a smart way, to view or to filter such modifications.

Of course some VCS can now auto-generate some of those information, and even provide a good color interface where one can see for each line who is the author. But, the comment in the file, the log, is put to insist; to insist that this modification is important. Not every commit is important to understand the code, but some are as they add a feature that has multiple implications on the code. Log comments are often very synthetic whereas commit message in VCS are usually very specific. A log comment is usually an agglomeration and summary of multiple commits. If those logs were more formal, it could help to better understand the code. One could see which parts of the code correspond to which added feature or optimizations, or even temporarily remove this feature or optimization to be able to better understand the original, simpler, code.

Firefox includes directly in his repository code from externally developed librairies. They also do a few modifications on those librairies and add comments to keep track of their changes.

```
 positionPtr = bufferPtr;
/* BEGIN MOZILLA CHANGE (always set eventPtr/eventEndPtr) */
  eventPtr = bufferPtr;
  eventEndPtr = bufferPtr;
/* END MOZILLA CHANGE */
  return result;
```

Chapter 4

Explanation()

The *Explanation* category allows the programmer to express his basic need to explain his code. This is the category most people associate with comments. A common wisdom is that comments are useful for program understanding and maintaining and so "explain code". But the term "explain code" is fuzzy. We have already seen many comments which are specialized forms of explanation, and that some could be supported by tools.

The explanation either repeat the code (which is bad), or summarize it by giving an higher-level idea. Note that the better the PL, the more the programmer can directly express high-level ideas in code and so does not need comments.

Tools like Javadoc have been invented to automatically extract the useful documentation from the comment. PL like LISP go even one step further by making the documentation, and some hypertext capabilities, part of the language (the docstring format of Lisp).

4.1 Example()

```
        rpc_gss_principal_t      client_principal;
!!      char     *svc_principal; /* service@server, e.g. nfs@caribe */
        rpc_gss_service_t        service;
```

4.2 Specific Explanations()

C constructs can be used for many things and sometimes programmers feel the need to put a comment to explain which one of the possibilities they use. Without this comment the reader can not fastly know visually by just looking at the construct what is the code for. For instance, if C provided only a control structure called iffor(), that can be used both for a if() and a for(), then programmers probably would add a comment each time to explain which one of the possibility they use. Fortunately, C provides both forms and so visually the programmer gets already lots of information by seeing the name of the keyword; his brain is better prepared for understanding the rest of the code. Providing a minimal set of general features is a good thing for a programming language, but adding the possibility to specialize them is also good.

- In C a loop can be used for many things: to iterate over numbers, over a list, a tree, to find an element in a list, to iterate over the elements, to erase some elements, etc.
- In C a `void*` can be used for many things
- In C bit operations can be used for many things

4.2.1 For Explanations()

A for() can be (ab)used to find an element, by using a goto or a break to stop the loop when the element is found. This is a common idiom [] (a kind of very basic design pattern []).

```
!!      /*
         * First, see if there is an active turnstile for the lock indicated
         * by the address.
         */
        lock = (struct lock_object *)addr;
        tc = TC_LOOKUP(lock);
        LIST_FOREACH(ts, &tc->tc_turnstiles, ts_hash)
                if (ts->ts_lockobj == lock)
                        goto found;
```

```
!!                      /* does dev_t in list match dev */
                        cnt = 0;
                        while (cnt < retndevs) {
                                if (*dev == md_expldev(ddi_devs[cnt]))
                                        break;
                                cnt++;
```

```
!!      /* Look for the translation */
        for (trans = table; trans->from; trans++)
                if (trans->from == from)
                        return trans;
```

Note that they use each time a different kind of code to provide a similar functionality. This hurts program understanding. If C provided a construct for_find(), then the programmer would not even have to understand the code; by just seeing for_find() he would understand that the goal is to find an element somewhere in the list.

In modern languages the programmer is not limited to basic control structures. He can extend in some way the language by introducing new control structures. One way is to provide iterators [], a way to loop easily over different kind of types. The other way is to define new kind of functions taking a special kind of parameter, some code, more precisely closures []. Those functions are called higher-order functions. So, the previous comments express the need in some way for such features. C++ now allows a basic form of closure. C# and Java also both support now closures. Most scripting languages like Python or Ruby allow iterators and closures.

```
// def
for_find(list, predicate) {
 for(x = list; x != NULL; x = x-> next) {
    if(code(x)) return x;
   }
}
// use
  xfound = for_find(mylist, fun(x) { x.size == 3 });
  // xfound point to the element in mylist having a size of 3
```

Such code is shorter than the original 'for', and as a consequence less buggy and more easy to understand. It requires the ability to provide code as-is, to give code as a parameter.

Linux programmers have realized the need to define new control structures and have (ab)used cpp macros to simulate that. For instance they use a special macro to iterate over a list instead of writing each time

the for. Unfortunately this can lead to mistakes as one has to take care to provide the good conditions each time:

```
/**
 * list_for_each        -       iterate over a list
 * @pos:        the &struct list_head to use as a loop cursor.
 * @head:       the head for your list.
 */
#define list_for_each(pos, head) \
        for (pos = (head)->next; prefetch(pos->next), pos != (head); \
                pos = pos->next)

/**
 * list_for_each_prev   -       iterate over a list backwards
 * @pos:        the &struct list_head to use as a loop cursor.
 * @head:       the head for your list.
 */
#define list_for_each_prev(pos, head) \
        for (pos = (head)->prev; prefetch(pos->prev), pos != (head); \
                pos = pos->prev)


// example of use
...
 list_for_each(x, mylist) {
   if(x.field == 1)
    break;
}
```

Here are other recurring comments concerning loops:

```
!!      /* Calibrated busy loop */
        while (count--> 0 && !(inb(iobase + UART_LSR) & UART_LSR_TEMT))
                udelay(1);
```

```
!!      /* busy wait */
        for (t = 0; t < 0x1000; t++) {
                if ((x = es_rd(es, ES1371_REG_SMPRATE, 4) & 0x00870000) ==
                    0x00000000)
                        break;
```

```
!!      /* poll until codec valid */
        for (i = 0; i < 1000; i++) {
                if (via_rd(via, VIA_AC97_CONTROL, 4) & VIA_AC97_CODEC00_VALID)
                        return (0);
                DELAY(1);
        }
```

```
!!      /* delete all destination addresses except the source */
        TAILQ_FOREACH(net, &stcb->asoc.nets, sctp_next) {
                if (net != src_net) {
```

```
!!              /* Seek to the end of the runlist. */
                while (rl->length)
                        rl++;
```

4.2.2 Bit Explanations()

4.2.3 List Explanations()

```
struct dev_info  {
        struct dev_info *devi_parent;   /* my parent node in tree       */
        struct dev_info *devi_child;    /* my child list head           */
        struct dev_info *devi_sibling;  /* next element on my level     */
        ...
}
```

4.3 Other specific explanations()

4.4 ShortNameExlain ()

A very important number of comments are in this category. The programmers seem to want to use short names for variables in the program, for fields, and especially for symbolic macro constant, to make it more compact to read, but at the same time feel the need to document in comment to what this short name correpond to. Maybe the IDE could use this information to provide a tooltip. Maybe an option could be used to switch between a short and long format when reading some code. A variable would have two possible names. This may be part of the PL.

```
  struct bus_options {
!!      u8      irmc;           /* Iso Resource Manager Capable */
        u8      cmc;            /* Cycle Master Capable */
        u8      isc;            /* Iso Capable */
        u8      bmc;            /* Bus Master Capable */
        u8      pmc;            /* Power Manager Capable (PNP spec) */
        u8      cyc_clk_acc;    /* Cycle clock accuracy */
```

```
!!      unsigned char   dp_esect;       /* end sector */
        unsigned char   dp_ehd;         /* end head */
```

```
  #define LCSR_IUU      0x00000080      /* Input FIFO Under-run Upper     */
!!                                      /* panel                          */
  #define LCSR_OOL      0x00000100      /* Output FIFO Over-run Lower     */
                                        /* panel                          */
  #define LCSR_OUL      0x00000200      /* Output FIFO Under-run Lower    */
                                        /* panel                          */
```

```
!!# define HD_TMC_1      0x55    /* time constant register chan 1 */
  #define HD_CMD_0      0x2c    /* command register chan 0, wo */
  #define HD_CMD_1      0x4c    /* command register chan 1, wo */
```

4.5 Ref()

Programmer use references to RFC, manuals, or websites to point to documentations that may be useful to understand the code.

```
  /*
   * This code is referd to RFC 2367
   */
```

```
!!/*
   * Definitions for ID TECH (www.idt-net.com) devices
   */
  #define IDTECH_VID              0x0ACD  /* ID TECH Vendor ID */
```

```
!!        /* Read the data via the internal pipeline through CDSN IO
             register, see Pipelined Read Operations 11.3 */
          MemReadDOC(docptr, buf, 1054);
```

```
!!/*
   * ISO/IEC 9899:1999
   * 7.18.3  Limits of other integer types
   */
  /* Limits of ptrdiff_t. */
  #define       PTRDIFF_MIN     INT32_MIN
  #define       PTRDIFF_MAX     INT32_MAX
```

```
 /* Improve fragment distribution and reduce our average
  * search time by starting our next search here. (see
  * Knuth vol 1, sec 2.5, pg 449) */
```

```
/* ...
 *  For more details look to AC '97 component specification revision 2.2
 *  by Intel Corporation (http://developer.intel.com) and to datasheets
 *  for specific codecs. ... */
```

But those informal description are not directly usable. It would be better to have a special annotation to crossref that can be directly clicked on to go to the relevant site (for paper documentation of course it may be harder). Maybe a rfc://, commit://, news://, blog://, mail://, file://, etc.

4.6 Diagram()

```
!!/*
   * Error log scratchpad register format.
   *
   * +---------+-----------------------+-----------+
   * |ASI_EIDR| PA to logging buf | # of err |
   * +---------+-----------------------+-----------+
   *  63     50 49                    6 5         0
   *
   */
```

```
   /* --------CMU1---------- */
   /* --CS0-----|--CS1------ */
   /* -H-|--L-- | -H- | -L-- */
```

Those ascii art diagrams could be used as-is to define types and accessors. In fact some domain specific languages like PADL [] are dedicated to the easy specification of formats (for network IP packet, or the CPU tables for virtual memory bit layout). Unfortunately they are not used by OS programmers. Note that ascii art diagrams have the benefit to be easily embedded in the code. They may be tedious to write but at least programmers can use regular text editors to understand the diagram.

An IDE could either help to visualize such diagrams, and/or provide plugins to help write them.

Literate programming [] can go very far in the ability to have complex drawing associated with the code as one can use the full power of TeX in comment (allowing to have TeX tabulars, or xypic pictures). But people don't use literate programming and the editors do not understand such special comment and so do not provide an easy way to see the diagram on the fly. One has first to compile with a special tool the source to be able to see the diagrams.

We found very few diagrams, except *Font* diagrams (explained later) which turns out to be mostly auto-generated from a program. We found very few ascii art diagrams in the sample. The CodeMap paper [] says that drawing diagrams is not well supported. This paper also shows that programmers to not stick to UML.

```
/* ...
* then the start of that hole will be the new head.  The
* simple case looks like
*        x | x ... | x - 1 | x
* Another case that fits this picture would be
*        x | x + 1 | x ... | x
* In this case the head really is somewhere at the end of the
* log, as one of the latest writes at the beginning was
* incomplete.
* One more case is
*        x | x + 1 | x ... | x - 1 | x
* This is really the combination of the above two cases, and
* the head has to end up at the start of the x-1 hole at the
* end of the log.
* ... */
```

The following comment is not about a diagram, but the comment refers to a *state* that must be interpreted in the context of a state machine described by a kind of diagram in a comment a few lines before in the source code.

```
        if (!is_newentry) {
                if ((!olladdr && lladdr != NULL) ||      /* (3) */
                    (olladdr && lladdr != NULL && llchange)) {  /* (5) */
                        do_update = 1;
                        newstate = ND6_LLINFO_STALE;
!!              } else                                   /* (1-2,4) */
                        do_update = 0;
        } else {
                do_update = 1;
                if (lladdr == NULL)                      /* (6) */
                        newstate = ND6_LLINFO_NOSTATE;
```

And here is the diagram:

```
        /*
```

```
         * newentry  olladdr   lladdr   llchange    (*=record)
         *       0        n        n         --      (1)
         *       0        y        n         --      (2)
         *       0        n        y         --      (3) * STALE
         *       0        y        y          n      (4) *
         *       0        y        y          y      (5) * STALE
         *       1        --       n         --      (6)   NOSTATE(= PASSIVE)
         *       1        --       y         --      (7) * STALE
         */
```

It is similar to a state machine. The programmer then manually encoded this machine via a set of optimal if/then/else and refers back to the original specification via comment annotations.

It would be better to let the programmer write directly in the language the state machine via some PL features, which would then be automatically compiled into an efficient set of if. In fact PL like OCaml provide such features, called pattern matching compilation. Another way would be to add to C some compile-time reflexion capabilities so that such extensions could be added without modifying gcc. Here is how the preceding state machine can be encoded in OCaml. Note that even '_' noted '__' in the comment is actually a feature of OCaml:

```
match newentry, olladdr, lladdr, llchange with
| 0, false, false, _ -> do_update = 0;
| 0, true, false, _ -> do_opdate = 0;
| 0, false, true, _ -> do_update = STALE;
...
```

4.7 Font()

An OS provides some terminal capabilities and as such has to deal with fonts.

```
/*-
 * This font lives in the public domain. It is a PC font, IBM encoding,
 * which was designed for use with syscons.
 */

const struct gfb_font bold8x16 = {
        8,
        16,
        {
        /* 6 */
        0x00,   /* ........ */
        0x00,   /* ........ */
        0x3c,   /* ..*****.. */
        0x60,   /* .**..... */
        0xc0,   /* **...... */
        0xc0,   /* **...... */
        0xfc,   /* ******.. */
        0xc6,   /* **...**. */
        0xc6,   /* **...**. */
        0xc6,   /* **...**. */
        0xc6,   /* **...**. */
        0x7c,   /* .*****.. */
        0x00,   /* ........ */
```

```
        0x00,    /* ........ */
        0x00,    /* ........ */
        0x00,    /* ........ */
  ...
}
```

Programmers even used different formats for font comments:

```
!!      0xfe, /* 11111110 */
```

```
!!0x40, /*  X      */
```

```
        0x10, /* 000 0000 */
```

Some (or maybe all) of those comments are generated by tools, which shows that comment can also be the target of a tool, not only its input. In the first comment the tool is named 'syscons'. The 0/1 variant is auto-generated from a very old AmigaOS program 'cpi2fnt' and some programmers on the linux mailing list actually complain that they wanted to modify the font but could not find and use the AmigaOS tool.

If C could provide an easier way to build directly in the source such complex "objects", then it would be also easier to modify them directly. This may be done again if C would provide some compile-time reflexion capability (like in MetaOCaml [], or maybe Xoc []) to enable the programmer to write the font in a convenient format (like `....**....`), which would then at compile-time be translated efficiently in the pair of integers (like `0x3, 0x3`) shows in the previous comment.

That may be one of the goal of Intentional Programming []: enabling the programmer to not manipulate source code as text but as a complex document, like a Word document, where one can embed spreadsheets, diagrams, around text, and for each of those embedded objects use the appropriate tool. Some research tools allow to add comments via voice or to associate videos to parts of code leading to multimedia "comments".

This need to build complex objects directly in the code, is also presents for pixmaps, but in this case they don't use comments but the initializer capability of C and strings.

```
/* XPM data of Open-File icon */
static const char * xpm_data[] = {
"16 16 3 1",
"       c None",
".      c #000000000000",
"X      c #FFFFFFFFFFFF",
"                ",
"   ......       ",
"   .XXX.X.      ",
"   .XXX.XX.     ",
"   .XXX.XXX.    ",
"   .XXX.....    ",
"   .XXXXXXX.    ",
"   .XXXXXXX.    ",
"   .XXXXXXX.    ",
"   .XXXXXXX.    ",
"   .XXXXXXX.    ",
"   .XXXXXXX.    ",
"   .XXXXXXX.    ",
"   .........    ",
"                ",
"                "};
```

4.8 Other

Many explanation comments also just explain code, put words, usually corresponding to high-level concept, instead of sometimes obscure macro name. There are lots of such comments.

4.8.1 Brief

```
                if (fs->fs_reclaim & FS_CHECKCLEAN)
!!                      /*
                         * nothing dirty was found in the buffer or inode cache
                         */
                        if ((isbusy == 0) && (isreclaim == 0) &&
```

```
!!      /* if there is a front file */
        if (cp->c_metadata.md_flags & MD_FILE) {
                if (fgp->fg_dirvp == NULL)
                        goto out;
```

4.8.2 Summary

```
!!/*
   * Read in the disk block containing the directory entry (dirclu, dirofs)
   * and return the address of the buf header, and the address of the
   * directory entry within the block.
   */
  int
  readep(pmp, dirclust, diroffset, bpp, epp)
```

```
!!/* convert leaf index to log2 leaf value */
  #define       LITOL2BSZ(n,m,b)        ((((n) == 0) ? (m) : cnttz((n))) + (b))
```

4.8.3 Long

```
!!/*
   * Build Transmit Segment Descriptors
   *
   * This function will take a supplied buffer chain of data to be transmitted
   * and build the transmit segment descriptors for the data.  This will include
   * the dreaded operation of ensuring that the data for each transmit segment
   * is full-word aligned and (except for the last segment) is an integral number
   * of words in length.  If the data isn't already aligned and sized as
   * required, then the data must be shifted (copied) into place - a sure
   * performance killer.  Note that we rely on the fact that all buffer data
   * areas are allocated with (at least) full-word alignments/lengths.
   *
   * If any errors are encountered, the buffer chain will be freed.
   *
   * Arguments:
```

```
   *     fup       pointer to device unit
   *     m         pointer to output PDU buffer chain head
   *     hxp       pointer to host transmit queue entry
   *     segp      pointer to return the number of transmit segments
   *     lenp      pointer to return the pdu length
   *
   * Returns:
   *     m         build successful, pointer to (possibly new) head of
   *               output PDU buffer chain
   *     NULL      build failed, buffer chain freed
   *
   */
  static KBuffer *
  fore_xmit_segment(fup, m, hxp, segp, lenp)
```

```
!!      /*
         * It is okay that we muck with the new unit here,
         * since no one else will know about the unit struct
         * until we commit it. If we crash, the record will
         * be automatically purged, since we haven't
         * committed it yet and the old unit struct will be found.
         */

        /* copy in the user's unit struct */
        err = ddi_copyin((caddr_t)(uintptr_t)mgp->mdp, (caddr_t)new_un,
            (size_t)mgp->size, mode);
        if (err) {
```

```
!!      /*
         *      sequential append at tail: append without split
         *
         * If splitting the last page on a level because of appending
         * a entry to it (skip is maxentry), it's likely that the access is
         * sequential. Adding an empty page on the side of the level is less
         * work and can push the fill factor much higher than normal.
         * If we're wrong it's no big deal, we'll just do the split the right
         * way next time.
         * (It may look like it's equally easy to do a similar hack for
         * reverse sorted data, that is, split the tree left,
         * but it's not. Be my guest.)
         */
        if (nextbn == 0 && split->index == sp->header.nextindex) {
```

```
!!/************************ EEPROM BANDS ****************************
   *
   * The iwl3945_eeprom_band definitions below provide the mapping from the
   * EEPROM contents to the specific channel number supported for each
   * band.
   *
   * For example, iwl3945_priv->eeprom.band_3_channels[4] from the band_3
   * definition below maps to physical channel 42 in the 5.2GHz spectrum.
```

```
   * The specific geography and calibration information for that channel
   * is contained in the eeprom map itself.
   *
   * During init, we copy the eeprom information and channel map
   * information into priv->channel_info_24/52 and priv->channel_map_24/52
   *
   * channel_map_24/52 provides the index in the channel_info array for a
   * given channel.  We have to have two separate maps as there is channel
   * overlap with the 2.4GHz and 5.2GHz spectrum as seen in band_1 and
   * band_2
   *
   * A value of 0xff stored in the channel_map indicates that the channel
   * is not supported by the hardware at all.
   *
   * A value of 0xfe in the channel_map indicates that the channel is not
   * valid for Tx with the current hardware.  This means that
   * while the system can tune and receive on a given channel, it may not
   * be able to associate or transmit any frames on that
   * channel.  There is no corresponding channel information for that
   * entry.
   *
   *****************************************************************/
```

```
!!/*
   * Notes on reference tracing on ill, ipif, ire, nce data structures:
   *
   * The current model of references on an ipif or ill is purely based on threads
   * acquiring a reference by doing a lookup on the ill or ipif or by calling a
   * refhold function on the ill or ipif. In particular any data structure that
   * points to an ipif or ill does not explicitly contribute to a reference on the
   * ill or ipif. More details may be seen in the block comment above ipif_down().
   * Thus in the quiescent state an ill or ipif has a refcnt of zero. Similarly
   * when a thread exits, there can't be any references on the ipif or ill due to
   * the exiting thread.
   *
   * As a debugging aid, the refhold and refrele functions call into tracing
   * functions that record the stack trace of the caller and the references
   * acquired or released by the calling thread, hashed by the structure address
   * in thread-specific-data (TSD).  On thread exit, ip_thread_exit destroys the
   * hash, and the destructor for the hash entries (th_trace_free) verifies that
   * there are no outstanding references to the ipif or ill from the exiting
   * thread.
   *
   * In the case of ires and nces, the model is slightly different. Typically each
   * ire pointing to an nce contributes to the nce_refcnt. Similarly a conn_t
   * pointing to an ire also contributes to the ire_refcnt. Excluding the above
   * special cases, the tracing behavior is similar to the tracing on ipif / ill.
   * Traces are neither recorded nor verified in the exception cases, and the code
   * is careful to use the right refhold and refrele functions. On thread exit
   * ire_thread_exit, nce_thread_exit does the verification that are no
   * outstanding references on the ire / nce from the exiting thread.
```

```
 *
 * The reference verification is driven from the TSD destructor which calls
 * into IP's verification function ip_thread_exit. This debugging aid may be
 * helpful in tracing missing refrele's on a debug kernel. On a non-debug
 * kernel, these missing refrele's are noticeable only when an interface is
 * being unplumbed, and the unplumb hangs, long after the missing refrele. On a
 * debug kernel, the traces (th_trace_t) which contain the stack backtraces can
 * be examined on a crash dump to locate the missing refrele.
 */
```

Chapter 5

Type()

The *Type* category allows the programmer to express his need to specify more semantic properties about some data-structures. People use comments for *Type* because the C type system is not expressive enough. It does not allow to define specific constraints on the set of values a variable can take. For instance the 'int' type is too general.

Many of the following comments can be supported by existing type-based annotation language or advanced PL (like Ada). So, in the following I will only indicate if the comment can *not* be supported by an existing language.

Those annotations can be used to statically check or in some case only dynamically check and can also used as an help for debugging at run-time.

A *Type* express a property on the value and in some sense can be seen as a specific case of the *Interface* category described later, but a type is more focused on the property of a value instead of a function. It is also a specific kind of invariant.

5.1 NULL()

People abuse pointers and NULL to represent an option (they should use instead an 'optional type' as in OCaml) and so then need special annotations to warn about such use. Pointers can be used for too many things and so the comment is here to describe which of those use of pointers the programmer use.

This leads to lots of bloated code, where tests for NULL are inserted at many places (the number of such tests may be really huge for OS code). Tools like Coverity try to warn about the missing of such tests, adding even more bloats, but a better solution would be to not require such test at all.

```
  /*
   * @param wrch          playback channel (optional; may be NULL)
   * @param rdch          recording channel (optional; may be NULL)
   * @param song          song name gets copied from here
   */
...
  static int
  dsp_oss_setsong(struct pcm_channel *wrch, struct pcm_channel *rdch, oss_longname_t *song
```

```
        struct page     *mr_page;       /* owning page, if any */
```

```
STATIC int                              /* error */
  xfs_bmap_add_extent_hole_delay(
        xfs_inode_t             *ip,    /* incore inode pointer */
```

```
        xfs_extnum_t            idx,    /* extent number to update/insert */
!!      xfs_btree_cur_t         *cur,   /* if null, not a btree */
```

```
  /*...
!! *    If ixlstp IS NOT null AND is not the first compiled ixl command and
   *    is not an ixl label command, returns an error.
   *    If ixlstp IS null, uses the first compiled ixl command (ixl_firstp)
   *    in place of ixlstp.
   *
   *    If no executeable xfer found along exec path from ixlstp, returns error.
   */
  int
  hci1394_ixl_set_start(hci1394_iso_ctxt_t *ctxtp, ixl1394_command_t *ixlstp)
  {
```

```
!! * Copy a list of attribute names into the buffer
!! * provided, or compute the buffer size required.
!! * Buffer is NULL to compute the size of the buffer required.
   */
  int
  ext4_xattr_list(struct inode *inode, char *buffer, size_t buffer_size)
  {
```

```
        } else {
                /*
                 * ...
!!              * Note that the ts_inheritor for the turnstile
                * may be NULL. If one exists, its t_prioinv
                * chain has to be updated.
                */
                ASSERT(ts->ts_waiters == 1);
                if (ts->ts_inheritor != NULL) {
```

```
!!      /*
         * Obtain the transaction wrapper and tw will be
         * NULL for the dummy and for the reclaim TD's.
         */
        if (...) {
                tw = ... Get_TD(old_td->hctd_trans_wrapper));
                ASSERT(tw == NULL);
```

Note that the NULL problem, the fact that pointers may or may not point to a valid memory, is also present in Java where we can have variables without binding, leading to the infamous NullPointerException. This would not happen if Java made a better design choice, like in OCaml, where every variables must have a binding, a value. C does not force the programmer to provide such a default value, as it may lead to some inefficiencies in very few cases where people don't want to initialize data (but note that many checkers like Valgrind considers most of those occurrences of code as buggy code, especially if the uninitialized data is then used as-is by the program). But C could provide such a possibility maybe by just changing the default behavior. C could force (by construction) the programmer to provide a value, and in some cases allow uninitialized variable but forcing the programmer to provide an explicit annotation. C could also

differentiate such variables by giving them a different type which would make it clear that it's a special kind of variable.

```
int x = 1;
int *p = &x;
int *q; // compiler error
int_unitialized *q;

int f(int *x) {
  ...
}
int f(int_unitialized *x) {
  ...
}
```

Then functions would never need to provide `/* assumes not NULL */` annotations as every variables and every pointers, would by default have a binding.

```
int f(int *x); /* assumes x not NULL */
```

Surprisingly we didn't find that many NULL comments, whereas it is a focus of multiple annotation languages and bug checker. Maybe because they were using previously such comments, to describe what they assume from the caller, but now instead add concrete tests in the code. Maybe Coverity had a influence (a bad one) on this aspect.

Ada 2005 has a 'not null' feature.

5.2 Bound()

Most comments expressing the condition on the bounds of arrays are added on parameters of functions. See the *Deputy* later below.

5.3 Range()

```
short charheight; /* lines per char (1-32) */
```

Such range data types can be handled by Ada which either statically prove the correctness of code or insert few dynamic checks for the places where the static analysis failed.

```
#define WSIZE 0x8000     /* window size--must be a power of two, and */
                         /*  at least 32K for zip's deflate method */
```

```
  #define RX_RING_SIZE          (FEC_ENET_RX_FRPPG * FEC_ENET_RX_PAGES)
!!#define TX_RING_SIZE          16       /* Must be power of two */
  #define TX_RING_MOD_MASK      15       /*   for this to work */
```

The preceding comment is not really a range but still an extra condition on the domain of the value. It is also at a cpp-level which again would make it hard to check by regular tools.

The constraint can also be on the format of a strng:

```
        rpc_gss_principal_t     client_principal;
!!      char    *svc_principal; /* service@server, e.g. nfs@caribe */
        rpc_gss_service_t       service;
  } rpc_gss_rawcred_t;
```

5.4 Unit()

Unit types, also known as dimension types [], have their origin in physics where programmers manipulate different values of different dimensions, such as speed, distance, time, that can be combined together to build value of other dimensions. For instance, a distance can be divided by a time and can then be compared with a speed value as they have the same dimension (the same type). Without dimension types, one can compare a speed value with a distance value, which does not make sense, but which can not be detected by type system like C where every value would have the same (too general) type: 'int'.

In OS code, programmers seem to use mainly

- time dimension. They sometimes use seconds, milliseconds, micro seconds (usec), nanoseconds, or weeks. They always use the same type, 'int', for all of this. Maybe timing error can be made.
- byte dimension, with either kilo-bytes, sectors, byte ranges, or words.

They sometimes also use an hertz dimension type and bandwidth dimension type.

```
!!            xge_os_mdelay( 50 ); // wait 50 milliseonds
```

It would be better to have instead:

```
xge_os_mdelay(50ms);
```

```
/*
 * The default maximum commit age, in seconds.
 */
#define JBD_DEFAULT_MAX_COMMIT_AGE 5
```

In fact they sometimes define macros for such purpose like

```
/* milli second, micro second, nano second */
#define MSsec * 1000000
#define USsec * 1000
#define NSec  * 1
delay(100 MSec);
```

But again, macros are error prone and not type-checked. Note that the delay(10ms), delay(10s), could be done via OO classes with different time constructors: delay(new Seconds(1)), delay(new Msec(1000)). But maybe we could have a more lightweight solution based on unit types.

Note that TeX forces programmers to add unit to numbers as in `2.8in`.

```
!!static long tick_delta_sum;  /* Accumulated tick difference (usec).*/
```

```
!!#define        RNG_RETRY_HLCHK_USECS              100000   /* retry every .1 seconds */
```

```
  #define        DQ_FTIMELIMIT    (7 * 24*60*60)          /* 1 week */
```

```
  /* ...
   *  Split timeout and the returned value are in bus
   *    cycles.
   */
  static uint_t
  hci1394_async_timeout_calc(hci1394_async_handle_t async_handle,
      uint_t current_time)
  {
```

```
!!              aic_pci_write_config(pci, PCIR_COMMAND, command, /*bytes*/1);
                ahc->bugs |= AHC_PCI_MWI_BUG;
```

```
                case 'K':
!!                      if (num > max_bytes / 1024) /* will overflow */
                                return (EINVAL);
```

```
!!      /* Check size */
        if (data_buffer_length > TW_MAX_IOCTL_SECTORS * 512) {
```

```
int mem;        /* memory in 128 MB units */
```

If we use annotation `/* @@unit: 128MB */`, or if the C language lets programmers specify a unit as an extended type, then we can perform type checking on units to detect such mistakes.

```
        unsigned char   dp_ssect;       /* starting sector */
        unsigned char   dp_shd;         /* starting head */
        unsigned short  dp_scyl;        /* starting cylinder */
!!      unsigned char   dp_esect;       /* end sector */
        unsigned char   dp_ehd;         /* end head */
        unsigned short  dp_ecyl;        /* end cylinder */
        unsigned char   dp_name[16];
  };
```

```
    sc->config.synth.n    = 52; /* 52.000 Mbs */
```

5.5 State type()

A variable can go through multiple states. Some functions accept only a variable when it is in a specific state. Protocols works like this. The sing# [] language allows to specify complex protocols and ensure that the code conforms to this protocol.

```
int f(int fd); /* must be opened */
```

But we didn't find such comment in the sample.

```
/**
   * Given a connected NdbMgmHandle, turns it into a transporter
   * and returns the socket.
   */
  NDB_SOCKET_TYPE connect_ndb_mgmd(NdbMgmHandle *h);
```

5.6 Region pointers()

```
register INT32 * bptr;  /* pointer into bestdist[] array */
```

```
JSAMPLE * cptr;         /* pointer into bestcolor[] array */
```

5.7 Dependent types()

Dependent types are used to specify that the *type* of a variable depends on the *value* of another variable. For instance, given a function taking an array of int, and a length, one would like to say that the type of the array is of a specific length:

```
int foo(int *xs /* array[length] */, int length);
```

With this type information the type-checker can then try to detect out-of-bounds array access. This can be very useful as an OS manipulates lots of resources, and usually bounded resources.

C++ allows a form of dependent type as one can give in the type of a template an integer as in `vector<10> x;`. But it does not use this opportunity to statically check out-of-bounds array access.

I think Pascal allows to have flexible-arrays where the size is known at run-time and checks are done dynamically.

```
  typedef struct xfs_dirent {           /* data from readdir() */
        xfs_ino_t       d_ino;          /* inode number of entry */
        xfs_off_t       d_off;          /* offset of disk directory entry */
!!      unsigned short  d_reclen;       /* length of this record */
        char            d_name[1];      /* name of file */
  } xfs_dirent_t;
```

5.7.1 Array dependent types()

```
/*
 * PARAMETERS:  AmlBuffer           - Pointer to the resource byte stream
 *              AmlBufferLength     - Size of AmlBuffer
...
 */
ACPI_STATUS
AcpiRsGetListLength (
    UINT8                   *AmlBuffer,
    UINT32                  AmlBufferLength,
    ACPI_SIZE               *SizeNeeded)
{
```

In the preceding comment an additional parameter is passed with an array to give the length of this array. We didn't find that much such comments whereas such "checks", for bound checking, is a hot topic among annotation languages. Maybe it is important, but we didn't find that many such comments. Maybe programmers don't use comment to document such a dependency.

5.7.2 Union dependent types()

```
  typedef struct {      /* Auxiliary vector entry on initial stack */
!!      long    a_type;                 /* Entry type. */
        union {
                long    a_val;          /* Integer value. */
                void    *a_ptr;         /* Address. */
                void    (*a_fcn)(void); /* Function pointer (not used). */
        } a_un;
```

```
    union {                              /* chunk memory handles */
!!              struct ib_mr     *rl_mr;      /* if registered directly */
                struct rpcrdma_mw { ... }    /* if registered from region */
    }
    ....
    u64          mr_base;          /* registration result */
```

Note that in the comment above the condition ("if registered from region") is fuzzy and may require thinking to map it to a code condition.

```
  struct au_generic_tid {
!!      uchar_t gt_type;          /* AU_IPADR, AU_DEVICE,... */
        union {
                au_ip_t           at_ip;
                au_port_t         at_dev;
        } gt_adr;
  };
```

```
        u_int8_t         ID_type;          /* Layer 3 protocol discriminator */
        union {                            /* Layer 3 protocol */
                u_int8_t         simple_ID;       /* ITU */
                u_int8_t         IPI_ID;          /* ISO IPI */
!!              struct {                          /* IEEE 802.1 SNAP ID */
                        u_int8_t         OUI[3];
                        u_int8_t         PID[2];
                } SNAP_ID;
                u_int8_t         user_defined_ID;/* User-defined */
        } ID;
```

The Deputy [] dependent type system can provide such annotations for union using the following syntax: `WHEN(tag ==0)`. The annotation describes on which condition the member of an union must be used.

```
struct foo
{
  int tag;
  union foo {
    int *p; WHEN(tag == 0);
    int n;  WHEN(tag == 1);
  } u;
};
```

Some functional languages like OCaml or Haskell with advanced abstract types can directly support such alternatives without needing to relate multiple variables together with tags. The tag is generated internally by the compiler.

```
type foo =
| Pointer of int *p
| Int     of int n
```

5.8 Relation types()

```
!!/* ARL Structure, one per link level device */
  typedef struct arl_s {
        struct arl_s    *arl_next;              /* ARL chain at arl_g_head */
        queue_t         *arl_rq;                /* Read queue pointer */
        queue_t         *arl_wq;                /* Write queue pointer */
```

```
                struct {
!!                      GUID birth_volume_id;   /* Unique id of volume on which
                                                   the file was first created.*/
                        GUID birth_object_id;   /* Unique id of file when it was
                                                   first created. */
                        GUID domain_id;         /* Reserved, zero. */
```

Note the use also of a range constraint "zero".

This notion of uniqueness may be related to SQL and relational database systems. In a database the system enforces such constraints. UML also allows to annotate relations between classes with arity constraints like 1-to-1, 1-to-n, etc.

5.9 Memory types()

An OS manages multiple address space: its own kernel space and the address space of different process. At some point I think some pointers were qualified in comments as referring to either kernel pointers or user pointers (`/* kernel space variable */`). The Sparse [] annotation language introduced specific annotations (__user, __kernel) to ensure that such pointers were used only through specific functions. This may explain why we didn't find any comment anymore about the address space property of a pointer.

```
!!/* Conversion to new PCI API :
   * Convert an address in a kernel buffer to a bus/phys/dma address.
   * This work *only* for memory fragments part of lp->page_vaddr,
   * because it was properly DMA allocated via pci_alloc_consistent(),
   * so we just need to "retrieve" the original mapping to bus/phys/dma
   * address - Jean II */
```

5.10 Bit and bytes()

The following comments show that the C language is in fact not low-level enough. It does not make it very easy to manipulate bits and bytes. The C language proposes bitwise operators (`&`, `|`, `^`) and shifting operators (`>>`, `<<`), but they are tedious to use for doing complex bits and bytes manipulations.

```
  #define IRASH             0x4E000000      /* mask for changeable attributes */
!!#define ATTRSHIFT         25      /* bits to shift to move attribute
                                     specification to mode position */
```

5.10.1 Bitset

```
!!/* Clock flags */
  #define RATE_CKCTL                (1 << 0)        /* Main fixed ratio clocks */
  #define RATE_FIXED                (1 << 1)        /* Fixed clock rate */
  #define RATE_PROPAGATES                   (1 << 2)        /* Program children too */
```

```
#define VIRTUAL_CLOCK          (1 << 3)        /* Composite clock from table */
#define ALWAYS_ENABLED                 (1 << 4)        /* Clock cannot be disabled */
```

C++ now provides a special template library for bitset that I guess offer better type guarantees (but template are harder to use than such simple cpp macros).

5.10.2 cpplint

As we have seen in previous sections, lots of comments are not attached to C entities but cpp entities like macros, and express some conditions on macros. Most static analysis tools work at the C level and so lose the opportunity to detect bugs due to misuse of cpp. Most tools get rid of comments and cpp, so they can't really find some opportunity, such as the #define grouping type opportunity presented later. They only "see" integers instead of symbolic constants coming from lots of different macros. A paper [] studied the different usage and bugs coming from the use of cpp []. So, it would be useful to invent a lint for cpp, a cpplint.

Macro variables like #define X 1, are used for many things: to represent a symbolic constant, to be part of a bitset, to represent a magic number, a bitmask, to be used to align bits, etc. There are lots of opportunities to misuse such variables, e.g., to use an align macro variable instead of a bitset macro variable.

Another paper [] proposed a new kind of pre-processor. This pre-processor is safer, and also makes it it easier to implement program-transformation tools, like refactoring tools over C, as the use of the actual cpp is one of the biggest barrier [**?**] for the development of such tools.

```
!!/*
   * __stringify doesn't likes enums, so use SOCK_DCCP (6) and IPPROTO_DCCP (33)
   * values directly, Also cover the case where the protocol is not specified,
   * i.e. net-pf-PF_INET-proto-0-type-SOCK_DCCP
   */
  MODULE_ALIAS_NET_PF_PROTO_TYPE(PF_INET, 33, 6);
  MODULE_ALIAS_NET_PF_PROTO_TYPE(PF_INET, 0, 6);
```

5.10.3 Group

```
!!/*
   * EATA Command & Register definitions
   */
  #define PCI_REG_DPTconfig                       0x40
  #define PCI_REG_PumpModeAddress                         0x44
 ...
```

```
!!/*
   * MBOX registers
   */
  #define HE_REGO_CS_STPER0       0x000
  #define HE_REGO_CS_STPER(G)     (HE_REGO_CS_STPER0 + (G))
```

```
!!/*
   * Supervisory LLC commands
   */
  #define         LLC_RR          0x01
  #define         LLC_RNR         0x05
  #define         LLC_REJ         0x09
```

There is no assurance that those macros are used on the good variables or that they get mixed up with orthogonal macros.

5.10.4 Devil

Here are some read/write specifications that could be leveraged by some tools:

```
!!# define   EHCI_CONF_CF            0x00000001 /* RW configure flag */

  #define EHCI_PORTSC(n)                    (0x40+4*(n)) /* RO, RW, RWC Port Status reg */
  ...
  #define   EHCI_PS_PP               0x00001000 /* RW,RO port power */
  #define   EHCI_PS_LS               0x00000c00 /* RO line status */
```

```
#define E1000_CTRL      0x00000  /* Device Control - RW */
#define E1000_CTRL_DUP  0x00004  /* Device Control Duplicate (Shadow) - RW */
#define E1000_STATUS    0x00008  /* Device Status - RO */
!!# define        E1000_MCC         0x0401C  /* Multiple Collision Count - R/clr */
  #define        E1000_LATECOL     0x04020  /* Late Collision Count - R/clr */
```

```
!! /* Clear any pending interrupt events. */
   E1000_WRITE_REG(hw, E1000_IMC, 0xffffffff);
   icr = E1000_READ_REG(hw, E1000_ICR);
```

```
        /* 0x0042 - 0x0047:      reserved */
  #define B0_XM2_ISRC             0x0048  /* 16 bit ro    XMAC 2 Interrupt Status Reg */
        /* 0x004a - 0x004f:      reserved */
  #define B0_XM2_PHY_ADDR 0x0050          /* 16 bit r/w   XMAC 2 PHY Address Register */
        /* 0x0052 - 0x0053:      reserved */
!!# define B0_XM2_PHY_DATA 0x0054          /* 16 bit r/w   XMAC 2 PHY Data Register */
        /* 0x0056 - 0x005f:      reserved */
```

Here are complex bit and bytes layout as well as conditions:

```
  #define        PCI_ASPM_FORCE_CLKREQ_ENA       BIT_4    /* Force CLKREQ Enable (A1b only) *
  #define        PCI_ASPM_CLKREQ_PAD_CTL BIT_3    /* CLKREQ PAD Control (A1 only) */
!!# define        PCI_ASPM_A1_MODE_SELECT BIT_2    /* A1 Mode Select (A1 only) */
  #define        PCI_CLK_GATE_PEX_UNIT_ENA       BIT_1    /* Enable Gate PEX Unit Clock */
```

```
  typedef enum {
!!        PMUGPIO_MBC,            /* Boston MBC FPGA GPIO - 8-bit */
          PMUGPIO_CPLD,           /* Seattle CPLD GPIO - 8-bit */
          PMUGPIO_OTHER           /* Chalupa - 8-bit */
  } pmugpio_access_type_t;
```

```
!!/* [RW 16] all producer and consumer of port 0 according to the following
      addresses; U_prod: 0-15; C_prod: 16-31; U_cons: 32-47; C_cons:48-63;
      Defoult_prod: U/C/X/T/Attn-64/65/66/67/68; Defoult_cons:
      U/C/X/T/Attn-69/70/71/72/73 */
  #define HC_REG_P0_PROD_CONS                                          0x108200
  /* [RW 16] all producer and consumer of port 1according to the following
      addresses; U_prod: 0-15; C_prod: 16-31; U_cons: 32-47; C_cons:48-63;
```

```
      Defoult_prod: U/C/X/T/Attn-64/65/66/67/68; Defoult_cons:
      U/C/X/T/Attn-69/70/71/72/73 */
```

```
  #define EB2__CAW_8 0x00000000       /* SDRAM external bank column address width = 8 bits =
  #define EB2__CAW_9 0x00100000       /* SDRAM external bank column address width = 9 bits =
!!# define EB2__CAW_10 0x00200000      /* SDRAM external bank column address width = 9 bits
  #define EB2__CAW_11 0x00300000      /* SDRAM external bank column address width = 9 bits
```

```
!!/* Timer registers */
  #define       HPET_TIMER_CAP_CNF(x)   ((x) * 0x20 + 0x100)
  #define       HPET_TCAP_INT_ROUTE     0xffffffff00000000
  #define       HPET_TCAP_FSB_INT_DEL   0x00008000
  #define       HPET_TCNF_FSB_EN        0x00004000
  #define       HPET_TCNF_INT_ROUTE     0x00003e00
```

```
  #define PIOD_ASR                (0xa00 + 112) /* Select A register */
!!# define PIOD_BSR                (0xa00 + 116) /* Select B register */
  #define PIOD_ABSR               (0xa00 + 120) /* AB Select status register */
  #define PIOD_OWER               (0xa00 + 160) /* Output Write enable register */
  #define PIOD_OWDR               (0xa00 + 164) /* Output write disable register */
```

```
        /* AC97_SINGLE("Digital Audio Mode", AC97_AD_MISC, 12, 1, 0), */ /* seems problema
        AC97_SINGLE("Low Power Mixer", AC97_AD_MISC, 14, 1, 0),
        AC97_SINGLE("Zero Fill DAC", AC97_AD_MISC, 15, 1, 0),
!!      AC97_SINGLE("Headphone Jack Sense", AC97_AD_JACK_SPDIF, 9, 1, 1), /* inverted */
        AC97_SINGLE("Line Jack Sense", AC97_AD_JACK_SPDIF, 8, 1, 1), /* inverted */
  };
```

```
  #define PERF_MON_CNTH_REG       regptr(MSP_SLP_BASE + 0x148)
!!                                        /* Perf monitor counter high    */
  #define PERF_MON_CNTL_REG       regptr(MSP_SLP_BASE + 0x14C)
                                          /* Perf monitor counter low     */
```

```
  #define IXGB_TPTH     0x02104 /* Total Packets Transmitted (High) */
!!# define IXGB_GPTCL    0x02108 /* Good Packets Transmitted Count (Low) */
  #define IXGB_GPTCH    0x0210C /* Good Packets Transmitted Count (High) */
  #define IXGB_BPTCL    0x02110 /* Broadcast Packets Transmitted Count (Low) */
```

```
  #define EMU_DOCK_MAJOR_REV    0x25    /* 0000xxx  3 bit  Audio Dock FPGA Major rev */
!!# define EMU_DOCK_MINOR_REV    0x26    /* 0000xxx  3 bit  Audio Dock FPGA Minor rev */

  #define EMU_DOCK_BOARD_ID     0x27    /* 00000xx  2 bits Audio Dock ID pins */
  #define EMU_DOCK_BOARD_ID0    0x00    /* ID bit 0 */
```

```
!!/*
   * Error log scratchpad register format.
   *
   * +--------+------------------+----------+
   * |ASI_EIDR| PA to logging buf | # of err |
   * +--------+------------------+----------+
   *  63    50 49                6 5        0
   *
   */
  #define ERRLOG_REG_LOGPA_MASK INT64_C(0x0003fffffffffc0) /* PA to log */
  #define ERRLOG_REG_NUMERR_MASK INT64_C(0x000000000000003f) /* Counter */
  #define ERRLOG_REG_EIDR_MASK INT64_C(0x0000000000003fff) /* EIDR */
```

```
  #define PCI_ROMBASE_MSK 0xfffe0000L /* Bit 31..17: ROM Base address */
  #define PCI_ROMBASE_SIZ (0x1cL<<14) /* Bit 16..14: Treat as Base or Size */
  #define PCI_ROMSIZE (0x38L<<11) /* Bit 13..11: ROM Size Requirements */
!! /* Bit 10.. 1: reserved */
```

```
!!/* RX Descriptor Base Low/High.
   *
   * These two registers store the 53 most significant bits of the base address
   * of the RX descriptor table.  The 11 least significant bits are always
   * zero.  As a result, the RX descriptor table must be 2K aligned.
   */
```

```
  #define CCM_REG_CCM_INT_MASK                                    0xd01e4
  /* [R 11] Interrupt register #0 read */
  #define CCM_REG_CCM_INT_STS                                     0xd01d8
!!/* [RW 3] The size of AG context region 0 in REG-pairs. Designates the MS
     REG-pair number (e.g. if region 0 is 6 REG-pairs; the value should be 5).
     Is used to determine the number of the AG context REG-pairs written back;
     when the input message Reg1WbFlg isn't set. */
```

The preceding comments could be better handled by the domain specific language (DSL) Devil [] which makes it easier to manipulate bit and bytes, to combine bit, and to do it in a safe way with an advanced specific type checker.

Here is an example of a Devil specification:

TODO

One may ask why Devil, or more generally DSLs didn't "make it" into the OS community. Multiple DSLs have been developed but none of them worked, be it for low-level bit manipulation with Devil [] for device drivers code, or for low-level byte format specification for network packets with PADL [] and Melange [] for network protocols code. In the case of Devil one may think that the problem does not exist anymore as devices have now simpler standard interface which does not require to play with ports. In the past maybe some drivers have to do everything with 2 ports which require lots of tricks to provide lots of functionality through 2 ports, but maybe now device can have lots of ports as some memory or past constraints have vanished. Another explanation is that OS programmers don't want to learn new languages and prefer to stick with C, even if the low-level bit and byte manipulation are not as easy as in some DSL. DSL have, on the opposite, been quite successful in another domain, also dealing with a big system: compilers. Yacc, Lex, Burg [] are often used by compiler programmers. But, compiler programmers, who are often language lovers in the first place, are certainly more inclined to learn new languages, including DSLs.

5.11 Polymorphism, template types()

```
  struct zfcp_unit {
        struct device          sysfs_device;    /* sysfs device */
!!      struct list_head       list;            /* list of logical units */
        atomic_t               refcount;        /* reference count */
        wait_queue_head_t      remove_wq;       /* can be used to wait for
```

This comment may have lead to the introduction of template in C++, a better way to type-check container structures like list, tree, or hash.

```
list<logical_unit> units;
```

There is very few such comments in our sample as Linux programmers rarely use the generic list_head structure. They instead redefine each time different list structures like struct list_node, struct list_cpu, which is tedious and lead to duplication of code or bad type checking if they want to factorize functionalities. Also when they use list_head, they don't use comments but instead encode the type of the list in the name of the variable (which are also not type-checked by the compiler to find errors). So, it is very easy to mix list of oranges and list of apples in C.

5.12 Shape

5.13 Abuse int(), Abuse string()

This section is not about the name of a specific category but a theme that often repeats in the previous categories. OS programmers use the 'int' type for many things (for dimensions, for range, for bitset, even for memory address), and as one can manipulate an int with many different kinds of operators in C (arithmetic, logical, or even use pointer arithmetic) it may be very easy to make mistakes.

Some annotations may help a little like the attribute `__attribute(bitwise)` or the bitwise annotation of Splint(?).

In fact a similar abusing problem happens with char* pointers, which are used to represent byte regions, string, filename, or directory names. In Java this problem has been partially solved by introducing different classes, first a real String class, but also a File class, Directory class, that makes it more difficult to mix up things. For instance if a function takes both a filename and a string, in C if one use char* for both parameters, then there is no way for the typechecker to check that the argument are in the right order at the caller site. In Java those kinds of mistakes can be detected.

This is similar to some of the problems that beginners have with the C library. Many C functions take 2 parameters of the same type, for instance strcpy(), but some of those functions take first the source and then the destination while other functions do the opposite (inconsistencies in the API). This lead to lots of mistakes as one has to remember the different conventions.

Abusing 'int' or 'string' have also some advantages, which is the reason why people use them: it does not require to know the name of different types (or classes) and it does not require to remember the name of the conversion functions (or to call any conversion function at all). It's a quick and dirty technique, which makes it appealing for programmers.

5.14 Not seen in comments

We didn't find anything about information flow or privacy annotations. Maybe because information flow is a very new topic; what OS programmers don't know, they can not write comments about it. This may show also a limitation of our approach; some programmer needs may not be learned from comments.

Chapter 6

Interface()

The *Interface* category allows the programmer to express his need to specify how entities should be used, usually via some semantic properties. How to correctly use a value, a structure, a function ? What can be assumed and what can not be assumed ? What are the responsibilities ? What the caller/callee are responsible for ? What is the contract ?

6.1 Pre conditions()

There are lots of works to specify pre-conditions and post-conditions (larch, JML, ESC, etc) on parameters, such as "this function must take an integer that is more than 10 to work". But we didn't find that many such comments. We found pre-condition comments, but not on parameters. The comments we found were more about higher level pre conditions. Maybe because an OS is about managing state and time, it is less "functionnal" and so an OS maybe need lesss pre and post conditions on parameters and return value.

```
  /*...
   * Note that we rely on the fact that all buffer data
   * areas are allocated with (at least) full-word alignments/lengths.
   *
  ...
  static KBuffer *
  fore_xmit_segment(fup, m, hxp, segp, lenp)
```

```
!!        /* This breaks if a hash table grows above 32MB
           */
          hash_scratch = ((vm_offset_t)th->th_hashtable) | ((vm_offset_t)(1<<th->th_shift));
```

```
!!/* must be called with netlink table grabbed */
  static void netlink_update_socket_mc(struct netlink_sock *nlk,
```

```
   * Convert an address in a kernel buffer to a bus/phys/dma address.
   * This work *only* for memory fragments part of lp->page_vaddr,
   * because it was properly DMA allocated via pci_alloc_consistent(),
   * so we just need to "retrieve" the original mapping to bus/phys/dma
   * address - Jean II */
  static inline dma_addr_t virt_to_whatever(struct net_device *dev, u32 * ptr)
```

```
   * required, then the data must be shifted (copied) into place - a sure
!! * performance killer.  Note that we rely on the fact that all buffer data
   * areas are allocated with (at least) full-word alignments/lengths.
   ...
  static KBuffer *
  fore_xmit_segment(fup, m, hxp, segp, lenp)
```

```
!!/*                                                                    */
  /* WARNING:  It's the responsibility of the caller to make sure there */
  /* is enough room in rs_buf for the basic RPC message "preamble".     */
  /* ------------------------------------------------------------------ */
*/
  static int
  ippr_rpcb_decoderep(fin, nat, rs, rm, rxp, ifsrpcb)
```

```
  /*...
   *    This function assumes the current context is stopped!
   *
  int
  hci1394_ixl_set_start(hci1394_iso_ctxt_t *ctxtp, ixl1394_command_t *ixlstp)
  {
```

```
        if (pool == NULL) {
!!              /* The -1 assumes caller has done a svc_get() */
                nrservs -= (serv->sv_nrthreads-1);
        } else {
```

```
/*
 * This must be called only on pages that have
 * been verified to be in the swap cache.
 */
```

Note that such pre-conditions may not be easy to formalize. They correspond to high level concepts that does not map to C code directly. Maybe researchers need to study this: see if we can find ways to describe those high-level properties easily.

6.2 InOut()

```
  /* ------------------------------------------------------------------ */
  /* Function:    frpr_ah                                               */
  /* Returns:     void                                                  */
!!/* Parameters:  fin(I) - pointer to packet information                 */
  /*                                                                    */
  /* Analyse the packet for AH properties.                              */
  /* The minimum length is taken to be the combination of all fields in the */
  /* header being present and no authentication data (null algorithm used.) */
  /* ------------------------------------------------------------------ */
  static INLINE void frpr_ah(fin)
  fr_info_t *fin;
```

```
  {
```

```
  /**
   * nolock_hold_lvb - hold on to a lock value block
   * @lock: the lock the LVB is associated with
!! * @lvbp: return the lm_lvb_t here
   *
   * Returns: 0 on success, -EXXX on failure
   */

  static int nolock_hold_lvb(void *lock, char **lvbp)
```

```
  /*
   * PARAMETERS:   AmlBuffer          - ...
   *               AmlBufferLength    - ...
!! *               SizeNeeded         - Where the size needed is returned
   */
ACPI_STATUS
AcpiRsGetListLength (
    UINT8                   *AmlBuffer,
    UINT32                  AmlBufferLength,
    ACPI_SIZE               *SizeNeeded)
{
```

```
  /*              dst(I)  - pointer to byte sequence to search                */
  /*              slen(I) - match length                                      */
  /*              dlen(I) - length available to search in                     */
```

```
  /* ------------------------------------------------------------------------ */
  /* Function:    fr_fixskip                                                  */
  /* Returns:     Nil                                                         */
!!/* Parameters:  listp(IO)    - pointer to start of list with skip rule      */
  /*              rp(I)        - rule added/removed with skip in it.          */
  /*              addremove(I) - adjustment (-1/+1) to make to skip count,    */
  /*                             depending on whether a rule was just added   */
void fr_fixskip(listp, rp, addremove)
frentry_t **listp, *rp;
int addremove;
{
```

Note in the last comment another case of abuse of int for the adjustment parameter.

Note that the preceding comments are mostly all from the same file. Because our tool currently analyzes each comment *block* in separation, there will be 6 comments in the preceding example, one for each line. They are not currently agglomerated. So, the sample is biased to more often list comments from this file.

Those comments are here mainly because C can not return multiple values (tuples), and so pointers are used for that purpose. Pointers can be used both to modify an argument and to pass more effectively big arguments. Then programmers need to specify which modality they use by writing the role of each argument (is it an input or output). Functional languages don't have this problem and have thus far more cleaner function interface, closer to mathematics. A function takes only input arguments and can return multiple values.

This notion of in and out about arguments is also present in Interface Definition Languages (IDLs) like Corba. Pascal and Ada also directly support in the language those annotations.

```
foo(in int x, out int y) return int
begin
 ...
end
```

C++ and C supports the 'const' qualifier which can be used to say what is 'in' and must not be modified. The absence of 'const' could be interpreted as an 'out' but this absence is sometimes due to sloppiness. Also, sometimes the parameter is both 'in' and 'out' in which case it can not be put as 'const'. Const alone can not fulfill the 3 possibilities which are 'in', 'out', and 'in out'. The splint [] annotation language can also support such annotations.

```
!!      struct ifqueue          inq;        /* queue of incoming mbuf's */
        struct ifqueue          outq;       /* queue of outgoing mbuf's */
  #define BT3C_DEFAULTQLEN      12          /* XXX max. size of out queue */
  };
```

The preceding annotations can be used for parameters, but not for the specification of fields in structure as in the comment above. For such comment can a tool also ensure that the in and out queues are used in the good way ?

6.3 Context()

The conditions on the context of a call can not be expressed easily in C because it requires reflexion capabilities, such as the ability to go through all the set of functions calling another function. Lisp for instance provides such a capability.

One can use global variables and `assert` to mimic such a need (setting the global in the caller function and checking it in the callee). But, maybe because of the complex control-flow in an OS (with interrupts), it may not work well. Or maybe there are too many entry points that would need at each place to modify this global variable which makes the whole process more difficult.

```
/* read a key's data (optional)
         * - permission checks will be done by the caller
         * - the key's semaphore will be readlocked by the caller
         * - should return the amount of data that could be read, no matter how
         *   much is copied into the buffer
         * - shouldn't do the copy if the buffer is NULL
         */
        long (*read)(const struct key *key, char __user *buffer, size_t buflen);
```

6.3.1 Context Lock()

Note in the following comments the variety of use of words to express locking requirements (must, assume, hold, acquire, etc).

```
/* Lock must be acquired on entry to this function. */
```

```
/* caller must hold instance lock */
```

```
  /*
   * ...
   * Assume we hold the lock.
   */
  static void
  fatm_check_heartbeat(struct fatm_softc *sc)
  {
        uint32_t h;
```

```
  /* Tell common.c that B channel has been closed. */
!!/* cs->lock must not be locked */
  static inline void gigaset_bchannel_down(struct bc_state *bcs)
  {
```

```
/* information on the codec itself, plus function pointers */
struct codec_info {
        ...
        /* called when pcm stream is opened, probably not implemented
         * most of the time since it isn't too useful */
        int (*open)(struct codec_info_item *cii,
                    struct snd_pcm_substream *substream);

        ...
!!      /* stop() is called after data is no longer pushed to the codec.
         * Note that stop() must be atomic! */
        int (*stop)(struct codec_info_item *cii,
                    struct snd_pcm_substream *substream);
```

```
  /* ------------------------------------------------------------------------ */
  /* Function:    fr_authflush                                                */
  /* Returns:     int - number of auth entries flushed                        */
  /* Parameters:  None                                                        */
!!/* Locks:       WRITE(ipf_auth)                                             */
  /*                                                                          */
  /* This function flushs the fr_authpkts array of any packet data with       */
  /* references still there.                                                  */
!!/* It is expected that the caller has already acquired the correct locks or */
  /* set the priority level correctly for this to block out other code paths  */
  /* into these data structures.                                              */
  /* ------------------------------------------------------------------------ */
  int fr_authflush()
  {
```

Those kinds of comments are supported by iComment []. They represent 0.5% of the total number of comments and when grouped with the caller/callee comments they may add-up to the 1% number written in the iComment paper.

Sparse [] also supports the __require, and __release annotations to describe the requirements or effects regarding locks of functions. Lock_lint also provide lots of annotations regarding locks:

```
// from lock_lint manual
```

```
NOTE(MUTEX_ACQUIRED_AS_SIDE_EFFECT(MutexExpr))
NOTE(READ_LOCK_ACQUIRED_AS_SIDE_EFFECT(RwlockExpr))
NOTE(WRITE_LOCK_ACQUIRED_AS_SIDE_EFFECT(RwlockExpr))
NOTE(LOCK_RELEASED_AS_SIDE_EFFECT(LockExpr))
NOTE(LOCK_UPGRADED_AS_SIDE_EFFECT(RwlockExpr))
NOTE(LOCK_DOWNGRADED_AS_SIDE_EFFECT(RwlockExpr))
NOTE(NO_COMPETING_THREADS_AS_SIDE_EFFECT)
NOTE(COMPETING_THREADS_AS_SIDE_EFFECT)
```

In the following the condition is not on the context before the call but on what must be done by the caller after the call.

```
  /* ------------------------------------------------------------------------ */
  /* Function:     ipf_findtoken                                              */
  /* Returns:      ipftoken_t * - NULL if no memory, else pointer to token    */
  /* Parameters:   type(I) - the token type to match                          */
  /*               uid(I)  - uid owning the token                             */
  /*               ptr(I)  - context pointer for the token                    */
  /*                                                                          */
  /* This function looks for a live token in the list of current tokens that  */
  /* matches the tuple (type, uid, ptr).  If one cannot be found then one is  */
  /* allocated.  If one is found then it is moved to the top of the list of   */
  /* currently active tokens.                                                 */
  /*                                                                          */
  /* NOTE: It is by design that this function returns holding a read lock on  */
!!/*       ipf_tokens.  Callers must make sure they release it!               */
  /* ------------------------------------------------------------------------ */
  ipftoken_t *ipf_findtoken(type, uid, ptr)
  int type, uid;
  void *ptr;
  {
    ...
        MUTEX_DOWNGRADE(&ipf_tokens);

        return it;
  }
```

In the following the comment is put at the call site as it is not easy from the name of the function to know that a side effect of the function is to release a lock:

```
!!        /* release the hold on the child */
          ndi_rele_devi(dip);
  ...
```

```
!!/**
   *  e1000_get_hw_semaphore_generic - Acquire hardware semaphore
   *  @hw: pointer to the HW structure
   *
   *  Acquire the HW semaphore to access the PHY or NVM
   **/
  s32 e1000_get_hw_semaphore_generic(struct e1000_hw *hw)
  {
```

```
!!/*
   * The audit_worker thread is responsible for watching the event queue,
   * dequeueing records, converting them to BSM format, and committing them to
   * disk.  In order to minimize lock thrashing, records are dequeued in sets
   * to a thread-local work queue.  In addition, the audit_work performs the
   * actual exchange of audit log vnode pointer, as audit_vp is a thread-local
   * variable.
   */
  static void
  audit_worker(void *arg)
  {
```

```
!!/*
   * Make sure the causing IRQ is blocked, then call do_IRQ. After that, unblock
   * and jump to ret_from_intr which is found in entry.S.
   *
   * The reason for blocking the IRQ is to allow an sti() before the handler,
   * which will acknowledge the interrupt, is run. The actual blocking is made
   * by crisv32_do_IRQ.
   */
  #define BUILD_IRQ(nr)                   \
  void IRQ_NAME(nr);                      \
  __asm__ (                               \
```

```
!!/*
   *    Recover an error report and clear atomically
   */
  static inline int sock_error(struct sock *sk)
  {
  ...
        err = xchg(&sk->sk_err, 0);
  ...
  }
```

```
/*
 * This function will acquire the lock and set the in_transition
 * bit for the specified slot.  If the slot is being used,
 * we return FALSE; else set in_transition and return TRUE.
 */
static int
sysc_enter_transition(int slot) { ... }
...
  /* mutex lock the whole list */
 if (sysc_enter_transition(-1) != TRUE) {
```

Programmers have extended their lock library to provide debugging and self-defense capabilities. Opensolaris provides the MUST_HELD+ macro.

```
/* ...
 * Assumes: tq->tq_lock is held.
 */
```

```
static void
taskq_ent_free(taskq_t *tq, taskq_ent_t *tqe)
{
      ASSERT(MUTEX_HELD(&tq->tq_lock));
```

Note that even with this macro, the programmer still felt the need to also put a comment about the locking requirment. Indeed the ASSERT is in the body of the code and the clients of this function usually read only the interface, that is the comment preceding the function.

```
/*
 * Move a page back to the lists.
 *
 * Must be called with the slab lock held.
 *
 * On exit the slab lock will have been dropped.
 */
```

```
        /*
         * no locking for this, because it does its own
         * plus, it does a kmalloc
         */
```

```
/*
 * We're allowed to run sleeping lock_page() here because we know the caller has
 * __GFP_FS.
 */
```

6.3.2 Context Caller()

The Java programming language provides a stack inspection mechanism [] to express conditions on the caller, but they are enforced at run-time only.

6.3.3 Context Interrupt()

```
/* this function must not be called from interrupt or completion handler */
```

```
!!/*
   * Free TX resources.
   *
   * Assumes that SGE is stopped and all interrupts are disabled.
   */
  static void free_tx_resources(struct sge *sge)
  {
```

```
!!/*
   * Callback for the Tx buffer reclaim timer.  Runs with softirqs disabled.
   */
  static void sge_tx_reclaim_cb(unsigned long data)
  {
```

```
        /*
          * Need to run this when irqs are enabled, because it wants
          * to self-test [hard/soft]-irqs on/off lock inversion bugs
          * too:
          */
```

A set of functions could contain in comments a tag to indicate, to mark, to which category they belong to, like `/* @contextcategory: interrupt && completion */`. Functions could then easily express requirements by adding in their own comment like `/* contextrequire: interrupt || completion */`.

6.3.4 Other context

Some context conditions can also be expressed at a "module" level instead of function level, giving in one comment a condition on a set of functions.

```
  /*
   * Operations on bitmaps of arbitrary size
   * A bitmap is a vector of 1 or more ulong_t's.
!! * The user of the package is responsible for range checks and keeping
   * track of sizes.
   */

  #ifdef _LP64
  #define        BT_ULSHIFT      6 /* log base 2 of BT_NBIPUL, to extract word index */
```

```
!!/*
   * Refresh the HAT ismttecnt[] element for size szc.
   * Caller must have set ISM busy flag to prevent mapping
   * lists from changing while we're traversing them.
   */
  pgcnt_t
  ism_tsb_entries(sfmmu_t *sfmmup, int szc)
  {
```

```
/*...
 *      For tight control over page level allocator and protection flags
 *      use __vmalloc() instead.
 */
```

```
/*
 * This function only removes the unlocked pages, if you want to
 * remove all the pages of one inode, you must call truncate_inode_pages.
 *
 * invalidate_mapping_pages() will not block on IO activity. It will not
 * invalidate pages which are dirty, locked, under writeback or mapped into
 * pagetables.
 ...
```

```
 /*
   * A simple loop like
```

```
  *     while ( jiffies < start_jiffies+1)
  *             start = read_current_timer();
  * will not do. As we don't really know whether jiffy switch
  * happened first or timer_value was read first. And some asynchronous
  * event can happen between these two events introducing errors in lpj
  * ... */
```

```
// Allocate a String in the Arena and register that String so that it is
// deallocated at the same time as the Arena.
// DO NOT CALL DELETE ON THE RESULT!
JS::String &JS::newArenaString(Arena &arena)
{
    String *s = new(arena) String();
    arena.registerDestructor(s);
    return *s;
}
```

```
/*
 * Macros to make the correct C datatypes for a new kind of ring.
 *
 * To make a new ring datatype, you need to have two message structures,
 * let's say struct request, and struct response already defined.
 *
 * In a header where you want the ring datatype declared, you then do:
 *
 *     DEFINE_RING_TYPES(mytag, struct request, struct response);
 *
 * These expand out to give you a set of types, as you can see below.
 * The most important of these are:
 *
 *     struct mytag_sring      - The shared ring.
 *     struct mytag_front_ring - The 'front' half of the ring.
 *     struct mytag_back_ring  - The 'back' half of the ring.
 *
 * To initialize a ring in your code you need to know the location and size
 * of the shared memory area (PAGE_SIZE, for instance). To initialise
 * the front half:
 *
 *     struct mytag_front_ring front_ring;
 *     SHARED_RING_INIT((struct mytag_sring *)shared_page);
 *     FRONT_RING_INIT(&front_ring, (struct mytag_sring *)shared_page,
 *                     PAGE_SIZE);
 *
 * Initializing the back follows similarly (note that only the front
 * initializes the shared ring):
 *
 *     struct mytag_back_ring back_ring;
 *     BACK_RING_INIT(&back_ring, (struct mytag_sring *)shared_page,
 *                    PAGE_SIZE);
 */
```

6.3.5 SmPL

The Semantic Patch Language [**?**]SmPL) could be used as an annotation language and programmers could embed in comments SmPL scripts to detect bad code at the caller site, in the context.

```
 /* @Smpl:
-   foo(...);
+   error(); printf("use foobar instead foo");
    ...
    bar(x)
 */
bar(int x)
{
 ...
}
```

SmPL could also make use of other annotations, to match over those annotations. This would require that SmPL understand comments.

```
@@
function f;
@@
/* ... @context: interrupt ... */
f(...) {
+ foo();
 ...
}
```

6.3.6 Buffer Ownership()

This category allows the programmer to specify *responsibilities*.

```
                if (mp_nce == NULL) {
!!                      /* The caller will free mp */
                        mutex_exit(&nce->nce_lock);
```

Note that I don't know what is 'mp'. Maybe this comment is obsolete.

```
  /* ...
   * It includes pre-registered buffer memory for send AND recv.
!! * The recv buffer, however, is not owned by this structure, and
   * is "donated" to the hardware when a recv is posted. When a
   * reply is handled, the recv buffer used is given back to the
   * struct rpcrdma_req associated with the request.  ... */
```

There are type systems, like ownership types [], that try to solve such problems. Note that we found very few examples of such comments.

6.4 Effects()

```
!!      /* reconfigure AGP hardware again */
        nvidia_configure();
```

```
!!      /* Fill SG Array with new values */
        ivtv_udma_fill_sg_array(dma, y_buffer_offset, uv_buffer_offset, y_size);
```

```
!!      pci_read_config_dword(tp->pdev, PCICOMMAND, &reg); /* flush write */
        udelay(100);
```

```
/* this function call exit(0) */
```

Again, a SmPL script could easily detect such case if relevant annotations were provided:

```
@rule1@
function f;
@@
/* ... @kind: exit ... */
f() { ... }

// look for exit labeled function
@@ rule.f @@
  f();
- S
+ { print dead code }
```

Some side effects are often described at the caller site. Indeed from the name of the function it is often difficult to know that the function have side effects. In fact language like Scheme use some conventions in the standard library to add a `?` or `!` to describe respectively predicate functions and functions with side effects as in `(is_digit? "45o")` and `(add_list! 1 xs)`. But those are only conventions. A checker could enforce that such visual clues are indeed true.

6.5 Error()

Programmers use different conventions to signal an error or success. They can use NULL when returning pointers (leading to NULL problems described in the *NULL* section), or an 'int' code like -1, or sometimes 0, or sometimes something else. They can use the same conventions to signal a success. Sometimes an error is a negative value, sometimes a positive value. As there is no PL mechanism to describe errors (like the exception mechanism in C++) and as each programmer can use his own variant, comments are used to specify the conventions for each functions. It is yet another case of abuse of 'int' and yet another use of comments to disambiguate situations because of the lack of enforced conventions.

Here is an interesting question asked on the kernel mailing list:

```
from http://www.ussg.iu.edu/hypermail/linux/kernel/0607.3/1252.html
> Hello,
>
> I'm looking at the source code of different drivers and wondering about
> request_irq() return value. It is used mostly in 'open' routine of struct
> net_device. If request_irq() fails some drivers return -EAGAIN, some -EBUSY
> and some the return value of request_irq(). Is this intentional? Sample
> drivers code:

Correct practice is to propagate the error code of request_irq out to be
the return value of the open routine. This allows the request_irq to return
```

```
different values for overlapping irqs, or out of memory, etc.

> Besides request_irq() is arch dependent so depending on arch it has different
> set of possible return values. So ... does the return value matter or I
> misunderstood something here?

Each architecture should return something sane. If it doesn't then it a problem
that should be addressed there.
```

```
!!/*
   * Do a transaction.
   * return 1 if ACK, 0 if NAK, -1 if error.
   */
  static int
  slhci_transaction(struct slhci_softc *sc, usbd_pipe_handle pipe,
        u_int8_t pid, int len, u_char *buf, u_int8_t toggle)
```

```
  /*
!! * Returns: 0 on success, -EXXX on failure
   */
  static int nolock_hold_lvb(void *lock, char **lvbp)
  {
```

```
   * Return value:
   *    target address on Success / 0 on Failure
   */
  static u64
  get_target_identifier(peidx_table_t *peidx)
  {
```

```
!!/*                                                                      */
  /* Returns:                                                           */
  /*   0 on success, positive value on failure.                         */
  /**********************************************************************/
  static int
  bce_nvram_test(struct bce_softc *sc)
```

```
  /*
   * @retval EINVAL      Operation not yet supported.
   */
  static int
  dsp_oss_setsong(struct pcm_channel *wrch, struct pcm_channel *rdch, oss_longname_t *song
```

```
  plat_cpu_poweron(struct cpu *cp)
  {
!!      return (ENOTSUP);       /* not supported on this platform */
  }
```

Note that in the last 2 examples the "not yet supported" is represented as different macros.

```
     if (na->acl.fattr4_acl_len != vs_ace4.vsa_aclcnt)
!!           error = -1; /* no match */
     else if (ln_ace4_cmp(na->acl.fattr4_acl_val,
```

```
/* ------------------------------------------------------------------------ */
/* Function:     memstr                                                     */
/* Returns:      char *  - NULL if failed, != NULL pointer to matching bytes */
/* Parameters:   src(I)  - pointer to byte sequence to match                */
/*               dst(I)  - pointer to byte sequence to search               */
/*               slen(I) - match length                                     */
/*               dlen(I) - length available to search in                    */
/*                                                                          */
/* Search dst for a sequence of bytes matching those at src and extend for  */
/* slen bytes.                                                              */
/* ------------------------------------------------------------------------ */
char *memstr(src, dst, slen, dlen)
const char *src;
char *dst;
size_t slen, dlen;
{
```

```
        if (pri == 0)
                goto done;
!!      if (pri > 0) {  /* error */
                scheme = &g_part_null_scheme;
                pri = INT_MIN;
        }
```

```
!!STATIC int                                    /* error */
  xfs_bmap_add_extent_hole_delay(
```

The need for this last comment is that functions can also return actual numeric values resulting from numerical computations. It would be better to have a clean error type to avoid such ambiguity, or to use exceptions. A possibility would be to introduce an enum type for errors, as now ANSI C compilers report as an error to use an int as an enum (there is no more implicit cast from and to enum).

Note that the use of signed vs unsigned should help gcc to detect errors. In practice it looks like gcc does not do much static checks on this issue. Moreover, because of the preceding error requirement, most numerical function returning a positive int use signed int instead of unsigned int to deal with errors. Errors thus prevent some checking and limit the usefulness of the specific signed type.

The lack of exception in C also lead either to code bloats, as any caller of a function must insert code each time to test for the return value and error code, or bugs, as some caller may forget to add such a test.

Note that the use of exceptions in the beginning also led to the introduction of comments (a new kind of comment), Programmers put in comments next to the signature of the function the set of exceptions that can be raised by the function, as it is indeed part of the interface:

```
int foo(int x); /* raise Not_Supported exn */
```

This comment in turn led to the introduction of a feature, the possibility to add in the interface those information:

```
int foo(int x) raise Not_Supported;
```

This information can be used, for instance in Java, to check that the code indeed raises such an exception and that each call sites or their parents handle at one point the situation. This is possible also in C++ but many programmers still put such error interface information in comments.

The OCaml language provide an advanced static tools called `ocamlexn` [] that detects if some exceptions are never "captured" and pop-up to the toplevel.

The lack of clean error types and exceptions and the impossibility to return multiple values makes the interface of functions "unnatural". Compare the C and Java version:

```
-----------------------------------------------------------------
// C ugly code, requires lots of comments to make it bug-resistant:
// - in/out comment,
// - buffer ownership comment,
// - error comment

/* -1 means that the input string is not in a good format */
int int_of_string(/* in */ char* s, /* out */ int *x) {
 ...
}
/* return error code. This function allocates the memory for bar */
int foo(char *s, /* out */ struct *bar) {
  int res;
  int error = int_of_string(s);
  if(error == -1) {
    printf("int not in good format");
    return error;
  }
  do_stuff;
  bar = malloc(...);
  bar.myint = 1;
  return 0; /* everything is ok */
}

-----------------------------------------------------------------
// Java code (better interface), and less error prone
// at caller site. No need for comment.
int int_of_string(String s) raise StringIntBadFormat {
 ...
}
Bar foo(String s) {
   int i = int_of_string(s);
   do_stuff;
   Bar res = new Bar();
   res.myint = i;
   return res;
}
```

The introduction of exceptions enabled to really use the return type of a function for the return value, which in turns made the use of in and out annotations useless. This makes the type of the function closer to a mathematical specification.

In fact this problem is mentionned also in the C++ Mozilla project at `http://weblogs.mozillazine.org/roadmap/archives/2006/10/mozilla_2.html`:

```
PRBool
nsXULDocument::OnDocumentParserError()
{
  // don't report errors that are from overlays
  if (mCurrentPrototype && mMasterPrototype != mCurrentPrototype) {
    nsCOMPtr<nsIURI> uri;
    nsresult rv = mCurrentPrototype->GetURI(getter_AddRefs(uri));
    if (NS_SUCCEEDED(rv)) {
      PRBool isChrome = IsChromeURI(uri);
      if (isChrome) {
        nsCOMPtr os(
          do_GetService("@mozilla.org/observer-service;1"));
        if (os)
          os->NotifyObservers(uri, "xul-overlay-parsererror",
                              EmptyString().get());
      }
    }
    return PR_FALSE;
  }
  return PR_TRUE;
}

you'll see code like this:

bool
XULDocument::OnDocumentParserError()
{
  // don't report errors that are from overlays
  if (mCurrentPrototype && mMasterPrototype != mCurrentPrototype) {
    IURI *uri = mCurrentPrototype->GetURI();
    if (IsChromeURI(uri)) {
      GetObserverService()->NotifyObservers(uri, "xul-overlay-parsererror");
    }
    return false;
  }
  return true;
}
```

6.6 Magic number()

Programmers should use types, variants, or at least symbolic constants to represent special conditions. If not, then they need to repeat the convention in comments at multiple places. The special case value is sometimes 0, sometimes -1, sometimes maxint.

```
!!      ld.ld_magic = 0;                    /* indicate end of messages */
        dumpvp_write(&ld, sizeof (ld));
```

```
!!      /* Reject application specific interfaces
         */
        if (hostif->desc.bInterfaceClass != 255) {
```

```
!!      /* Set defaults for nTxLock and nTxBlock if unset */
        if (nTxLock == -1) {
```

```
  /*
   * A value of 0xff stored in the channel_map indicates that the channel
   * is not supported by the hardware at all.
   *
   * A value of 0xfe in the channel_map indicates that the channel is not
   * valid for Tx with the current hardware.  This means that ... */
```

```
!!      /* OFDM rates */
        case 12:
        case 18:
                return 12;
```

```
                spin_lock(&cadet_io_lock);
!!              outb(7,io);             /* Select tuner control */
                outb(curvol,io+1);
```

```
                        case 0x096: /* Lens cursor */
                        case 0x097: /* Intuos3 Lens cursor */
                                wacom->tool[idx] = BTN_TOOL_LENS;
                                break;
!!                      case 0x82a: /* Eraser */
                        case 0x85a:
```

```
                lxpnp->lxpr_mode = 0500;        /* read-search by owner only */
                break;
  ...
        case LXPR_NETDIR:
                vp->v_type = VDIR;
                lxpnp->lxpr_mode = 0555;        /* read-search by all */
                break;
```

```
        if (dma > 3 || dma < 0 || dma == 2) {
          ...
        } else {
!!              /* Extended mode DMA enable */
                cfg = 0x50;

                if (dma == 3) {
                        dma_bits = 3;
                } else {
```

```
!!        /*
           * IR == -1 if DA < DB, IR == 0 if DA == DB,
           * IR == 1 if DA > DB and IR == 3 if unorderded
           */
          emu_set_CC(regs, (IR == -1) ? 1 : (IR == 1) ? 2 : IR);
```

```
  void do_syscall_trace(void)
  {
!!        /*
           * The 0x80 provides a way for the tracing parent to distinguish
           * between a syscall stop and SIGTRAP delivery
           */
          ptrace_notify(SIGTRAP|((current->ptrace & PT_TRACESYSGOOD) ? 0x80 : 0));
```

For those comments, the only thing a tool could do is to try to detect them and warn the user that he should define a symbolic constant, or an enum. Some tools like ArgoUML [] try to apply AI technique on source code or model, in order to report bad design choices.

6.7 Module interface()

The C language does not have a real module system. Instead programmers use cpp and #include directives to achieve more or less what a module system can provide. C does not know about cpp; cpp does not know about C. cpp just "see" text, which has some advantages as the generic #include mechanism can be (ab)used for other things than module handling (for instance to factorize parts of code). But it has also some disadvantages. Comments are used to try to incorporate some of the advanced features of module system in other PL.

```
#include <xxx.h> // pdflush_operation()
```

```
#include <linux/buffer_head.h>  /* for try_to_release_page(),
                                   buffer_heads_over_limit */
```

```
#include <netinet/in.h> /* For in6_addr_t */
#include <sys/tsol/label.h> /* For brange_t */
#include <sys/tsol/label_macro.h> /* For brange_t */
```

PL like Haskell or Perl provide advanced module import/export mechanisms. One can easily states which function he wants to use from a module. If this function is not used anymore then the compiler could warn the user who could delete the then useless import (actually I think those modern PL do not provide such a warning). This is harder with cpp as most tools, again, do not work at the cpp level.

It's yet another kind of comment related to the use and limitations of cpp. Many of the features of C++ can in fact be traced to the desire to replace the use of cpp by real PL features. In this case C++ namespace. C++ inline helped avoiding using macros, C++ const helped avoiding defining constant via macros, etc.

```
!!/*
   * For netgraph nodes that are somehow associated with file descriptors
   * (e.g., a device that has a /dev entry and is also a netgraph node),
   * we define a generic ioctl for requesting the corresponding nodeinfo
   * structure and for assigning a name (if there isn't one already).
   *
   * For these to you need to also #include <sys/ioccom.h>.
   */

  #define NGIOCGINFO     _IOR('N', 40, struct nodeinfo)  /* get node info */
  #define NGIOCSETNAME   _IOW('N', 41, struct ngm_name)  /* set node name */
```

On the opposite to the first example, because one header can be used to provide signatures for multiple files, one may add in comment also where to find the implementation of a function prototype:

```
  /* tls.c */
  extern int os_set_thread_area(user_desc_t *info, int pid);
  extern int os_get_thread_area(user_desc_t *info, int pid);

!!/* umid.c */
  extern int umid_file_name(char *name, char *buf, int len);
```

```
        .flush_buffer     = ircomm_tty_flush_buffer,
!!      .ioctl            = ircomm_tty_ioctl,     /* ircomm_tty_ioctl.c */
        .tiocmget         = ircomm_tty_tiocmget, /* ircomm_tty_ioctl.c */
        .tiocmset         = ircomm_tty_tiocmset, /* ircomm_tty_ioctl.c */
        .throttle         = ircomm_tty_throttle,
```

```
!!extern const int st_ndrivetypes;       /* defined in st_conf.c */
  extern const struct st_drivetype st_drivetypes[];
  extern const char st_conf_version[];
```

The last two examples illustrate the problem of the flat namespace of the C language. There is no easy way, seeing an entity, to know where it is defined. With C++ or other PL you can use namespaces to solve this problem by *qualifying* variables as in:

```
St_conf::st_ndrivetypes;
```

An IDE could also colorize differently the functions depending on their provenance to help the user understand the code.

```
!!/*
   * This is the loadable module wrapper.
   */
  #include <sys/modctl.h>
```

6.8 Time and Space properties()

```
!!/*
   * The TCP normal data output path.
   * NOTE: the logic of the fast path is duplicated from this function.
   */
  static void
  tcp_wput_data(tcp_t *tcp, mblk_t *mp, boolean_t urgent)
  {
```

Some functions in Linux are described as the slow or fast path. The fast being a specialized version of the slow or normal one. A profiler could check such claims (if those functions were clearly annotated) that indeed a version is faster than the other one. The iComment paper cites a paper on performance assertion [].

```
       /*
        * Merge cpu freelist into freelist. Typically we get here
        * because both freelists are empty. So this is unlikely
        * to occur.
        */
```

NOT IN SAMPLE

```
/* swapoff spends a _lot_ of time in this loop! ...*/
```

6.9 Other interface()

```
/* Must not sleep. */
static void
t1_config(softc_t *sc)
  {
```

Chapter 7

Code Relationships()

The *Core Relationships* category allows the programmer to understand how things work together. As opposed to the *Type* and *Interface* categories, here the programmer don't want to understand something in isolation but instead how an entity interacts with the other entities. Even if programmers try hard to isolate functionalities, to separate concerns, so things can be developed and understood separately, at one point functions or data-structures need to work together.

Unfortunately, as a human has only 2 eyes, when he looks at one place he can see only this place. A comment can help to see other places without really seeing them by describing those other places. This notion of focus+context is an important theme in the domain of information retrieval and data visualization. There are better ways to offer focus+context than using comments. An IDE could provide an annotation-based guided navigation capability where a programmer looking at some code, with special annotations, could be offered automatically the contextual information that is described by the annotation (for instance via a tooltip, or by reorganizing the source code view with some advanced fish-eye 2D effect). The IDE could be more pro-active in helping the programmer to understand the code. Note that [] describes that programmers spend a very significant portion of their time navigating in the code. Some annotations may help.

Note that there are already lots of implicit code relationships in the code (some functions calling other functions, or using some data structures), that tool can also leverage. But as there is so many such relations, the programmer usually use comments to *insist* on some of those relations, the most important one, and mark them. One could also maybe infer those important relations, for instance if a function calls multiple functions, maybe some of them are more important because less used (for install a call to kmalloc() is less important, because less original, than a call to ext2_helper_func()).

Code relation annotations can be used both for

- checking; check that the relation indeed holds
- code understanding; an editor and navigation tool can help the programmer by using those relation information.

In some way many of the diagrams in UML allow to better understand code relationships. But such relations are specified at the UML level and are generally not present at the code level. C++ can not express arity conditions for instance.

A program is a very complex mathematical objects with lots of possible relations:

- how data structures work together, type relations, how multiple fields or variables work together
- how functions work together, what are the protocols
- how control flows
- how data flows, from which variables specific value come from,

- pointer relations
- header vs implementation relations, files or modules relations, how files use other files
- callback relations
- how high-level concepts maps to concrete implementation
- how all of this mix together

7.1 File organization()

7.1.1 Visual organization()

Programmers often use rudimentary mechanisms to navigate in a file, like scrolling, and so add some visual clues in the file, some organizational delineates marks, to make it faster when scrolling to spot and separate the different parts of the file.

```
!!/*****************************************************************************
   *
   * RSDP - Root System Description Pointer (Signature is "RSD PTR ")
   *
   *****************************************************************************/
```

```
!!/* ------------------------ EEPROM UTILITIES: */

  /* Offsets of data in the EEPROM */
  #define EEPROM_COPYRIGHT        (0)
```

```
!!/* ---------------------------------------------------------------------------- */

  static inline struct xencons_interface *xencons_interface(void)
```

```
%#pragma mark ------ Local type definitions ------
```

Programmers have invented very sophisticated tools to visualize complex data for physicians, statisticians, but still visualize their own most important data, the source code, with very basic editors (as flat text files), and use very basic navigation mechanisms. For instance I also use in this latex file some visual markers to help me organize and navigate in my latex file.

Some editors now use some colors to mark different entities (function in blue, macros in yellow, etc) helping to better understand the code structure thanks to those visual hints. Some editors, like Source Insight [], also use different fonts to put in bigger font the header of function for instance. But those tools are still quite basic. An exception may be Code Thumbnails [] which proposes a "map" of the code that can be zoomed in and out, with colors, and allows the programmer to click on this map. This tool relies on the visual memory capability of the human. SoftViz [] provides a similar functionnality.

Note that programmers may have different taste concerning visualization, and Emacs for instance uses special comments (again) at the end of the file allowing the programmer to set some special variables conditioning the layout of the code.

```
int main() {
 ...
}
/*
```

```
 * Local Variables:
 * mode:c
 * comment-column:0
 * comment-start: "/**"
 * comment-end:"*/"
 * c-basic-offset: 8
 * End:
 */
```

Emacs also uses special comments at the beginning of the file (showing again that comments can be the artifacts that tool can rely on to store meta information).

```
/* -*-mode: c; fill-column: 75; comment-column: 50; -*- */
/* foo.c
 *
 */
int main()
{
```

```
/*--       --    - -           -           ---   ---
  \ \     / / -(-) |- ---   | |- ---   |    \|  - )
   \ \/\/ /  '-| |   -/ -_) |   -/ - \ | |) |  - \
    \-/\-/|-|  |-|\--\---|   \--\---/ |---/|---/ */
```

7.1.2 Grouping()

Another related use of comments, often used as a visual hint too, is to mark different entities as related by putting them together under a "section".

```
!!/* Debugging routines. */
  static char *get_elf_p_type(Elf32_Word p_type)
  ...
```

```
!!      /* Scan commands and notifications */
        REPLY_SCAN_CMD = 0x80,
        REPLY_SCAN_ABORT_CMD = 0x81,
```

```
!!      /* Input stuff. */
        struct string *prompt;          /* Output string for input area. */
        struct string *input;           /* Input string for read request. */
```

```
!!/*
   * control and status registers access macros
   */
  #define CSR_READ_1(sc, reg)                                          \
        bus_space_read_1((sc)->sc_st, (sc)->sc_sh, (reg))

  #define CSR_READ_2(sc, reg)                                          \
        bus_space_read_2((sc)->sc_st, (sc)->sc_sh, (reg))
```

```
  MODULE_LICENSE("Dual MPL/GPL");

!!/* Module parameters */
```

Note that those meta-information, about the correlation between different entities put next to each other in the file, is not used by any tool. Maybe one could check that those entities are indeed correlated in practice, and indeed used together. If not, this may indicate a bug or a bad grouping that may hinder program understanding.

```
 /* debugging function */
...
 /* helper function */
...
 /* globals */
...
 /* prototypes */
...
```

7.2 EndOfXXX()

Some closing constructs in C are ambiguous, like '}', as they can be used to close many different kind of statements (loops, if, switch). Programmers often feel the need to disambiguate those cases by adding a comment:

```
                rs->sn, rs->card_size);
!!      } /* if (!pcram_build_region_lists(rs)) */
```

```
        default:
                break;
!!      }; //Switch
  }
```

Some PL like Pascal or Ada actually enforce such disambiguations by having special constructs, which also enable to check that the closing annotation is correct (it can avoid bugs related to dangling else).

```
if(x = 1) do
  ...
end if

for (x = 1 to 4) do
 ...
end for
```

Some editors also provide balancing capabilities allowing to fastly know to which statement a closing statement corresponds to (and vice versa). Some editors also allow to automatically insert such comment when the programmer start to type the start of the statement.

In OS code the *EndOfXXX* comments are mainly used with cpp constructs like ifdef as in those cases there is no indentation information or easy balancing information that can be used to visually disambiguate. It's yet another example that shows how cpp constructs are badly supported, be it by checkers, or editors.

```
!!#else /* SHA2_UNROLL_TRANSFORM */
```

```
!!#endif                                /* defined(_FIRMWARE_NEEDS_FDISK) */
        {"a,raw", 0, S_IFCHR},
```

```
  #define SWI_IMBrange  0xf00001

!!#endif /* !_MACHINE_SWI_H_ */
```

Note that those kinds of ifdef (as well as endif), are used in headers (.h) to deal with the fact that C does not provide a real module system; people use cpp tricks to simulate a module system, which in turn also require some comments as cpp can be used for many things. This is in fact a recurring theme. C provide few constructs that can be used, and abused, for many different purposes; programmers then feel the need many times to specify which one of those uses the construct is used for. Maybe a bigger set of constructs would be better as the programmer would clearly see each time from the name of the construct the specific use.

```
!!#endif /* CONFIG_PPC_EARLY_DEBUG_44x */
```

```
!!#endif /* DEBUG */
```

In fact gcc and cpp do not provide much checks about ifdef. When Linus Torvalds wrote the Sparse tool, he wrote also a C pre-processor and found that many ifdefs were in fact not closed. I don't know if dangerous bugs can come from such miss.

Such comments are also used to indicate the end of the file (I never really understood its use) like:

```
/* this is the end of tls.c */
```

```
void Dbdih::gcpsavereqLab()
{
  sendLoopMacro(GCP_SAVEREQ, sendGCP_SAVEREQ);
  cgcpStatus = GCP_NODE_FINISHED;
}//Dbdih::gcpsavereqLab()
```

7.3 Control Flow()

In some way, the UML sequence diagram may help for the control flow part.

7.3.1 Caller Callee()

```
!!/* * Init CPU info - get CPU type info for processor_info system
call. */ void init_cpu_info(struct cpu *cp) {
```

```
/*
 * Line specific close routine, called from device close routine
 * and from ttioctl at >= splsofttty().
 * Detach the tty from the ppp unit.
 * Mimics part of tty_close().
 */
static int
pppclose(tp, flag)
```

```
!! * used by: simba_detach() on suspends
   *
   */
  static void
  simba_save_config_regs(simba_devstate_t *simba_p)
  {
```

```
!!/*
   * cvc_bbsram_start()
   *     Allow accesses to BBSRAM, used by cvc_assign_iocpu() after
   *     BBSRAM has been mapped to a virtual address.
   */
  static void
  cvc_bbsram_start(void)
  {
```

```
/*
 * Called from cache_reap() to regularly drain alien caches round robin.
 */
```

```
        /* instantiate a key of this type
         * - this method should call key_payload_reserve() to determine if the
         *   user's quota will hold the payload
         */
        int (*instantiate)(struct key *key, const void *data, size_t datalen);
```

7.3.2 Before After()

Programmer feel the need to express the context of a call, what happens before and after to indicate where this function fits in the general architecture.

```
/*
 * Initialisation.  Called after the page allocator have been initialised and
 * before smp_init().
 */
```

It could be useful to have a timeline view in the IDE. Also, the profiler could gives lots of information to help understand the program. There are lots of latent information during runs of a program that could be leveraged to help understand and check properties.

```
 /* Called before configuring an on-chip UART.  */
void ma_uart_pre_configure (unsigned chan, unsigned cflags, unsigned baud)
{
```

```
  /*
   * During init, we copy the eeprom information and channel map
   * information into priv->channel_info_24/52 and priv->channel_map_24/52
   *
   */
```

```
!!extern int ibmphp_init_devno (struct slot **);        /* This function is called from EBI
  extern int ibmphp_do_disable_slot (struct slot *slot_cur);
```

```
                        if (rval != DEVICE_RESET) {
                                rval = COMMAND_DONE_ERROR;
                        } else {
!!                              /*
                                 * Returning DEVICE_RESET will call
                                 * error recovery.
                                 */
                                severity = SCSI_ERR_INFO;
                                break; /* don't set position invalid */
                        }
```

```
/*
 * Note that it's ok to delay the TLB shootdown till the entire range is
 * finished, because if hat_pageunload() were to unload a shared
 * pagetable page, its hat_tlb_inval() will do a global TLB invalidate.
 */
        l = mmu.max_page_level;
        if (l == mmu.max_level)
```

```
!!                      /*
                         * Even though vp was obtained via vn_open(), we
                         * can't call vn_close() on it, since lofs will
                         * pass the VOP_CLOSE() on down to the realvp
                         * (which we are about to use). Hence we merely
                         * drop the reference to the lofs vnode and hold
                         * the realvp so things behave as if we've
                         * opened the realvp without any interaction
                         * with lofs.
                         */
                        VN_HOLD(lsp->ls_vp);
                        VN_RELE(vp);
```

```
/**
     * Called by nsIconProtocolHandler after it creates this channel.
     * Must be called before calling any other function on this object.
     * If this method fails, no other function must be called on this object.
     */
    NS_HIDDEN_(nsresult) Init(nsIURI* aURI);
```

7.3.3 Other

```
/* this is the main entry */
```

```
!!/*
   *     Main IP Receive routine.
```

```
 */
int ip_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt, struct n
{
        struct iphdr *iph;
        u32 len;
```

Programmers feel the need to annotate the main entry point of a module so that other programmers can know where to start when they want to understand the code. A source code visualizer could highlight such entry point if an annotation was present.

```
/*
 * Strange swizzling function only for use by shmem_writepage
 */
```

7.3.4 Unreached()

```
default: /* not reached */
```

```
        if (smb_mbc_encode(&sr->reply, fmt, ap) != 0) {
                va_end(ap);
                smbsr_encode_error(sr);
!!              /* NOTREACHED */
        }
        va_end(ap);
```

```
if (ip_sock == INVALID_SOCKET)
    {
      DBUG_PRINT("error",("Got error: %d from socket()",socket_errno));
      sql_perror(ER(ER_IPSOCK_ERROR));              /* purecov: tested */
      unireg_abort(1);                              /* purecov: tested */
    }
```

cf http://forge.mysql.com/wiki/DGCov_doc. Like FALLTHRU, used to shut down the default test coverage patch validator.

7.3.5 ProblematicControl() and FALLTHRU()

Some default behaviors of C are not always a good choice, especially for beginners. For instance in switch statements, the lack of a 'break' in a 'case' is often the sign of a bug. The same is true for the use of '=' instead of '==' inside an if. Those are syntactically correct control structures that are likely to be buggy. But, sometimes they are not buggy and this would be a false positive; a comment is then used to express this:

```
                case -EXDEV:                            /* partial completion */
                        gig_dbg(DEBUG_ISO, "%s: URB partially completed",
                                __func__);
!!                      /* fall through - what's the difference anyway? */
                case 0:                                 /* normal completion */
```

A specific annotation FALLTHRU or FALLTHROUGH is often used, mainly in OpenSolaris, to formally express those conditions. From the Lint manual [] `/* FALLTHRU */` or `/* FALLTHROUGH */`: "Suppress

complaints about fall through to a case or default labeled statement. This directive should be placed immediately preceding the label".

Note that there is a strict condition on where the annotation must be put for the tool to be able to grab it.

```
                        }
!!                      /* FALLTHRU */
                default:
                        IOSRAM_PUTB(tunnel, (uint8_t *)buf,
```

```
!!                      /* FALLTHRU */
                case LEN:       /* i: get length/literal/eob next */
                        j = c->sub.code.need;
```

7.3.6 Else Explanation()

This comment just repeat, usually in simpler terms, the condition in the corresponding if that may be far away.

```
                        if (link_state != BMSR_LSTATUS) {
                                /* link down again */
                                ...
                        } else {
                                /* link stays up */
                                if (slave->delay == 0) {
                                        if (!have_locks)
                                                return 1;
```

7.4 Data Flow()

Those comments are about the flow of value, not about the type of those values.

```
        timeout_id_t    msd_timeout_id; /* id returned by timeout() */
!!      bufcall_id_t    msd_reioctl_id; /* id returned by bufcall() */
        bufcall_id_t    msd_resched_id; /* id returned by bufcall() */
        int     msd_baud_rate;  /* mouse baud rate */
```

```
        DONTCARE(tcp->tcp_xmit_hiwater);        /* Init in tcp_init_values */
!!      DONTCARE(tcp->tcp_timer_backoff);       /* Init in tcp_init_values */
        DONTCARE(tcp->tcp_last_recv_time);      /* Init in tcp_init_values */
        tcp->tcp_last_rcv_lbolt = 0;
```

```
!!                  /* used by trees.c: */
      /* Didn't use ct_data typedef below to suppress compiler warning */
      struct ct_data_s dyn_ltree[HEAP_SIZE];   /* literal and length tree */
      struct ct_data_s dyn_dtree[2*D_CODES+1]; /* distance tree */
```

```
        int             nat_rev;                /* 0 = forward, 1 = reverse */
!!      int             nat_redir;              /* copy of in_redir */
  } nat_t;
```

```
        unsigned                first_offset;   /* offset into mapping[first] */
        unsigned                last_to;        /* amount of mapping[last] */
!!      unsigned short          offset;         /* offset into received data store */
        unsigned char           unmarshall;     /* unmarshalling phase */
```

```
!!      ibtl_cq_impl_flags_t    cq_impl_flags;  /* dynamic bits if cq */
                                                /* handler runs in a thread */
        int                     cq_in_thread;   /* mark if cq handler is to */
                                                /* be called in a thread */
```

```
  int zsasoftdtr = 0;   /* if nonzero, softcarrier raises dtr at attach */
  int zsb134_weird = 0; /* if set, old weird B134 behavior */
!!int g_zsticks = 0;    /* if set, becomes the global zsticks value */
  int g_nocluster = 0;  /* if set, disables clustering of received data */
```

```
        hrtime_t        vdev_last_try;  /* last reopen time              */
!!      boolean_t       vdev_nowritecache; /* true if flushwritecache failed */
```

The Spark Ada [] language allows to express advanced data-flow properties like describing how a value must "derives" from other variables (like the other parameters or global variables), and only from those variables, and checks if the implementation actually does this and only this. It also helps to understand the program by knowing from where a complex value come. It is a sort of assert on data-flow properties. It is like being able from the PL to interact with external static analysis data-flow checkers.

```
// Ada code, in the following count correspond to a global variable
procedure (int X, int *Y)
 /* Y derives from X and count */
begin
 ...
end
```

This can useful for the interface of functions but also for fields in structure, to explain how a field is filled in respect to other parts of the code.

```
struct dev_info  {
    ...
        int             devi_pm_dev_thresh;     /* "device" threshold */
    ...
}
```

```
int xxx = 12; /* threshold for wait time */
```

In the preceding comments a tool could check that the variables are used only as a threshold, that is used with specific comparison operators in an expression ("less than" C operator).

```
int files_given; /* if this is zero, use stdin */;
```

```
!!/*
   * Mailbox message types, for use in mboxsc_putmsg() and mboxsc_getmsg() calls.
   * NOTE: Clients should not use the MBOXSC_NUM_MSG_TYPES value, which
   *       is used internally to simplify future code maintenance.
```

```
   */

  #define         MBOXSC_MSG_REQUEST      0x01
  #define         MBOXSC_MSG_REPLY        0x02
  #define         MBOXSC_MSG_EVENT        0x04
```

```
/*
 * key under-construction record
 * - passed to the request_key actor if supplied
 */
struct key_construction {
```

```
        /* default payload length for quota precalculation (optional)
         * - this can be used instead of calling key_payload_reserve(), that
         *   function only needs to be called if the real datalen is different
         */
        size_t def_datalen;
```

7.4.1 Unused() and ARGSUSED()

Those comments are not really used to describe a relationship but a lack of relationship.

```
        unsigned int x:2;       /* unused bits */
```

```
unsigned short   closing_wait2; /* no longer used... */
```

```
#define ACE_WORD_SWAP_BD        0x04            /* not actually used */
```

Because this preceding comment annotate a cpp level entities, tools can not check the claim in the comment.

```
        int     aio_lio_opcode;         /* LIO opcode */
!!      int     aio_reqprio;            /* Request priority -- ignored */
        struct  __aiocb_private _aiocb_private;
  } oaiocb_t;
```

```
        int     histcounter_type; /* size (in bits) and sign of HISTCOUNTER */
!!      int     spare[2];       /* reserved */
  };
```

```
        size_t  pr_locked;      /* pages of locked memory */
!!      size_t  pr_pad;         /* currently unused */
        uint64_t pr_hatpagesize; /* pagesize of the hat mapping */
```

```
/*LINTED table used in scsb.o and system utilities*/
static uchar_t scb_10_fru_offset[] = {
```

```
!!/*
   * If this module needs a periodic handler for the interrupt distribution, it
   * can be added here. The argument to the periodic handler is not currently
```

```
   * used, but is reserved for future.
   */
  static void
  apic_post_cyclic_setup(void *arg)
  {
  _NOTE(ARGUNUSED(arg))
        /* cpu_lock is held */
```

A specific annotation ARGUSED is often used, mainly in OpenSolaris to more formally express those conditions. From the Lint [] manual: `/* ARGUSEDn */`: "Makes lint check only the first n arguments for usage; a missing n is taken to be 0 ..."

```
!!/* ARGSUSED */
  int
  t1394_free_addr(t1394_handle_t t1394_hdl, t1394_addr_handle_t *addr_hdl,
      uint_t flags)
  {
```

```
!!/*ARGSUSED3*/
  static int
  ses_ioctl(dev_t dev, int cmd, intptr_t arg, int flg, cred_t *cred_p, int *rvalp)
  {
        ses_softc_t *ssc;
```

Note that lint use a different angle on annotations. Those annotations are used not to help the tool to find bugs, but instead to shut-down lint to not report false positives, to not generate a warning about unused args. The same was true with the FALLTHRU annotation described before.

7.5 Other code-data correlations()

When two pieces of code need to work together, and when the modification of one such piece must entail the modification of the other piece, there is a *coupling*. Programmers try to avoid coupling, as one wants to separate concerns as much as possible so local modifications do not entail a massive reorganization of the source code. Nevertheless, it is hard to avoid coupling. In such cases, it is also hard to get support from the PL to enforce coupling. It requires non-local reasoning and working at the C meta-level, which C does not permit (but Lisp can).

```
/*...
 * Note: The descriptor_type and Type fields must appear in the identical
 * position in both the struct acpi_namespace_node and union acpi_operand_object
 * structures.
 */
```

```
 /*
 * NOTE: If you change the size of this eachproc structure you need
 *       to change the definition for EACH_QUAD_SIZE.
 */
```

7.5.1 DataClump()

Some variables must sometimes work together. This is called by Martin Fowler [] the data-clump bad smell (because it's a bad practice, programmers should gather those variables in a separate class).

```
...
static unsigned insize;  /* valid bytes in inbuf */
static unsigned inptr;   /* index of next byte to be processed in inbuf */
static unsigned outcnt;  /* bytes in output buffer */
...
        insize = 0;             /* valid bytes in inbuf */
        inptr = 0;              /* index of next byte to be processed in inbuf */
        outcnt = 0;             /* bytes in output buffer */
```

```
!!      u_int64_t xmitPackets;          /* number of packets xmit */
        u_int64_t xmitOctets;           /* number of octets xmit */
        u_int64_t recvPackets;          /* number of packets received */
        u_int64_t recvOctets;           /* number of octets received */
  };
```

```
  struct compstat {
!!      u_int32_t unc_bytes;      /* total uncompressed bytes */
        u_int32_t unc_packets;    /* total uncompressed packets */
        u_int32_t comp_bytes;     /* compressed bytes */
        u_int32_t comp_packets;   /* compressed packets */
```

```
        caddr32_t               cm_param;       /* mech. parameter */
!!      size32_t                cm_param_len;   /* mech. parameter len */
  } crypto_mechanism32_t;
```

```
  typedef struct xfs_attr_leaf_name_local {
        __be16  valuelen;               /* number of bytes in value */
        __u8    namelen;                /* length of name bytes */
!!      __u8    nameval[1];             /* name/value bytes */
  } xfs_attr_leaf_name_local_t;
```

```
!!      /* Input stuff. */
        struct string *prompt;          /* Output string for input area. */
        struct string *input;           /* Input string for read request. */
        struct raw3270_request *read;   /* Single read request. */
        struct raw3270_request *kreset; /* Single keyboard reset request. */
        unsigned char inattr;           /* Visible/invisible input. */
```

7.5.2 StructInitialize()

When fields are related, the programmer sometimes must initiliaze all of them at the same time. In such case, to make the difference with field tuning, the programmer put a comment before a set of related assignements.

```
        /*
         * setup parameter status
         */
        pcon.pc_len = SMT_MAX_INFO_LEN ;        /* max para length */
        pcon.pc_err = 0 ;                       /* no error */
        pcon.pc_badset = 0 ;                    /* no bad set count */
        pcon.pc_p = (void *) (smt + 1) ;        /* paras start here */
```

We should then check that programmers don't forget to set a field. C++ solves this problem by introducing the constructor concept.

7.5.3 Lock variables correlations()

Comments are used to describe in a structure which fields must be protected by which lock as OS code use fine-grained locking.

```
/* update a key of this type (optional)
         * - this method should call key_payload_reserve() to recalculate the
         *   quota consumption
         * - the key must be locked against read when modifying
         */
        int (*update)(struct key *key, const void *data, size_t datalen);
```

```
struct us_data {
        /* The device we're working with
         * It's important to note:
         *    (o) you must hold dev_mutex to change pusb_dev
         */
        struct mutex            dev_mutex;      /* protect pusb_dev */
        struct usb_device       *pusb_dev;      /* this usb_device */
```

```
  typedef struct dmu_buf_impl {
!!      /*
         * The following members are immutable, with the exception of
         * db.db_data, which is protected by db_mtx.
         */
```

```
!!      /*
         * Journal tail: identifies the oldest still-used block in the journal.
         * [j_state_lock]
         */
        unsigned long           j_tail;
```

```
[frame=single]
/* Sequence number for this transaction [no locking] */
tid_t t_tid;
```

```
  typedef struct scsa2usb_cpr {
        callb_cpr_t             cpr;            /* for cpr related info */
        struct scsa2usb_state   *statep;        /* for scsa2usb state info */
!!      kmutex_t                lockp;          /* mutex used by cpr_info_t */
  } scsa2usb_cpr_t;
```

```
!!/**
   * nolock_hold_lvb - hold on to a lock value block
   * @lock: the lock the LVB is associated with
   * @lvbp: return the lm_lvb_t here
   *
   * Returns: 0 on success, -EXXX on failure
```

```
   */

  static int nolock_hold_lvb(void *lock, char **lvbp)
```

```
!!/**
   * struct reference - TIPC object reference entry
   * @object: pointer to object associated with reference entry
   * @lock: spinlock controlling access to object
   * @data: reference value associated with object (or link to next unused entry)
   */

  struct reference {
        void *object;
        spinlock_t lock;
```

```
/*-
 * Locking key to struct socket:
 * (a) constant after allocation, no locking required.
 * (b) locked by SOCK_LOCK(so).
 * (c) locked by SOCKBUF_LOCK(&so->so_rcv).
 * (d) locked by SOCKBUF_LOCK(&so->so_snd).
 * (e) locked by ACCEPT_LOCK().
 * (f) not locked since integer reads/writes are atomic.
 * (g) used only as a sleep/wakeup address, no value.
 * (h) locked by global mutex so_global_mtx.
 */
struct socket {
        int     so_count;               /* (b) reference count */
        short   so_type;                /* (a) generic type, see socket.h */
        short   so_options;             /* from socket call, see socket.h */
                struct  mbuf *sb_sndptr; /* (c/d) pointer into mbuf chain */
                u_int   sb_sndptroff;   /* (c/d) byte offset of ptr into chain */
                u_int   sb_cc;          /* (c/d) actual chars in buffer */
```

The preceding comment shows great informal annotations.

Lock_lint can provide such functionality, but arguably with more tedious annotations:

```
// from lock_lint (solaris) manual
mutex_t lock1;
struct foo {
        mutex_t lock;
        int mbr1, mbr2;
        struct {
                int mbr1, mbr2;
                char* mbr3;
        } inner;
        int mbr4;
};
NOTE(MUTEX_PROTECTS_DATA(lock1, foo::{mbr1 inner.mbr1}))
NOTE(MUTEX_PROTECTS_DATA(foo::lock, foo::{mbr2 inner.mbr2}))
NOTE(SCHEME_PROTECTS_DATA("convention XYZ", inner.mbr3))
```

```
!!/*
   * Access to this queue is synchronized by the free page queue lock.
   */
  static TAILQ_HEAD(, vm_reserv) vm_rvq_partpop =
                                TAILQ_HEAD_INITIALIZER(vm_rvq_partpop);
```

```
/*
 * Protects updates to hugepage_freelists, nr_huge_pages, and free_huge_pages
 */
static DEFINE_SPINLOCK(hugetlb_lock);
```

A comment can also be used to say that a variable *does not* need to be correlated to a lock.

```
        int free_touched;                   /* updated without locking */
```

Again, as this relation is at the struct definition level, C can't express those kind of invariants. It requires reflexivity

the MUVI [] tool tries to detect such correlations.

Maybe a better alternative to those comments would be to group related variables inside a monitor []. That way access to those variables would be automatically protected without requiring any extra checking. Why OS programmers don't use monitors ? Because they are more heavyweight than locks which are more quick and dirty.

```
 /*
 * Hashtable for mapping Object keys to int values. The methods of this
 * hashtable are not synchronized, and if used concurently must be externally
 * synchronized
 */
```

7.5.4 Protocol()

```
/*
 * Close a cache and release the kmem_cache structure
 * (must be used for caches created using kmem_cache_create)
 */
```

```
 /*
 * Note that this function only works on the kmalloc_node_cache
 * when allocating for the kmalloc_node_cache.
 */
```

If a formal annotation describing which functions must be used with which function (or which deallocator to use after using a specific allocate), bugs could be found. For instance one can enforce that the programmer do not call directly free but instead the appropriate wrapper.

```
/* @source: */
int kmalloc_node_cache();

/* @sink: kmalloc_node_cache() */
int free_node_cache();
```

Such protocols can be inferred by works from Dawson Engler [] or the PR-miner [] tool. But it may still be better if programmer could provide formally such semantic information.

7.6 Repeat()

Programmers sometimes prefer to repeat some code, especially data structure definitions, and put it in comments, so that they don't need to have to switch between two places. This means that the navigation and visualization capabilities of their editors, or their knowledge of it, are poor. Note that this is a dangerous practice as there is nothing that enforces that the commented copy of the definitions is updated when the original is updated. The *CloneCode* category has the same problem, and may also benefit from special annotations, as described later.

Those kinds of comments may be obsolete and corresponds to very old code (to an older programming era), now that modern programmers can have multiple windows and even multiple screens. But, even with those features, advanced IDE or editors do not always make it useful or easy to efficiently use the provided screen space.

```
        asy->asy_cflag &= ~CBAUD;
!!      if (asy->asy_bidx > CBAUD) {     /* > 38400 uses the CBAUDEXT bit */
                asy->asy_cflag |= CBAUDEXT;
```

7.6.1 Repeat type()

Programmers sometimes repeat the typedef definition or structure definition.

```
compat_caddr_t ptr; /* unsigned char* */
```

In fact we didn't find the typedef repetition in OS (but found some in Mozilla) maybe because for instance Linus Torvalds advocates strongly against the use of typedefs for most cases (see the Linux coding style document [**?**]).

```
  int
  ext2_inactive(ap)
!!        struct vop_inactive_args /* {
                struct vnode *a_vp;
                struct thread *a_td;
        } */ *ap;
  {
        struct vnode *vp = ap->a_vp;
```

An IDE can help by providing a tooltip to show the definition of the structure or typedef alias. The Intellisense [] (intelligent completion) editor feature is also now very often used to assist programmers to choose which field of a structure or which method they want to use. With this feature, the programmer does not have to look at the documentation of the API, or navigate to the header file, or remember the name of those fields and methods.

7.6.2 Repeat parameters()

```
                if (mode_buf != NULL) {
                        scsi_mode_sense(csio,
!!                                      /*retries*/4,
                                        probedone,
```

```
                error = bus_dma_tag_create(
                                /* parent       */ NULL,
                                /* alignment    */ 1,
```

```
                                    /* boundary     */ 0,
!!                                  /* lowaddr      */ ADV_EISA_MAX_DMA_ADDR,
                                    /* highaddr     */ BUS_SPACE_MAXADDR,
                                    /* filter       */ NULL,
                                    /* filterarg    */ NULL,
                                    /* maxsize      */ BUS_SPACE_MAXSIZE_32BIT,
                                    /* nsegments    */ ~0,
```

```
        if ((ccb->ccb_h.status & CAM_DEV_QFRZN) != 0)
                cam_release_devq(ccb->ccb_h.path,
!!                                  /*relsim_flags*/0,
                                    /*reduction*/0,
                                    /*timeout*/0,
                                    /*getcount_only*/0);
```

This comment also shows yet another time the problem of using 'int' for everything, and here maybe also the need for unit type.

There are 2 solutions that can fulfill the need to know the parameter of a function, a tool-based one and language-based one. The tool-based one, present in IDE like Eclipse, allows to automatically ,when putting the cursor on the name of the function, to see in a tooltip the prototype of the function (and so of its arguments). The language-based one, called labeled argument [], or keyword argument, or named argument, is present in language like Smalltalk or OCaml. It allows at the call site to specify the name of the argument as in:

```
int *p;
int *q;
strcpy(dest:p, src:q);
//strcpt(src:q, dest:p); is also valid.
```

This feature also allows to give the arguments in any order.

Another solution, when a function has a very long list of argument, is to introduce a structure representing the arguments.

```
struct arg_foo = {
 int timeout;
 int* parent;
 ...
};

int main() {
 struct arg_foo x = {
   .timeout = 1;
   .parent = new(1);
 foo(x);
};
```

7.7 Designator()

C allows to construct complex structure or array via initializers. But, the first version of this feature was not good enough and programmers added comments to make things clearer. Later a gcc extension called *designator* was provided that almost makes such comment useless. This shows again that maybe the repeated

use of specific comments by programmers inspired a new programming feature. Maybe we can trace many PL features as improvements over comments. Maybe the full history of PL and software engineering was to turn comments into something that tools could use (we have of course no proof of that).

7.7.1 DesignatorField()

```
struct sess session0 = {
        &pid0,   /* s_sidp */
        0,       /* s_lock */
        ...
};
```

```
        {
                { T_DIRECT, SIP_MEDIA_FIXED, seagate, "ST39140*", "*"},
!!              /*quirks*/0, /*mintags*/2, /*maxtags*/2
        },
        {
```

```
!!      -1,                         /* .max_rretries      [Note 3]
*/
        -1,                         /* .max_wretries      [Note 3]
*/
        {0x44, 0x44, 0x46, 0x46}, /* .densities          Density codes [Note 1]
*/
```

The need to build complex value as-is may be related to the *Font* category where one want to build a complex font object but use comments for that purpose.

Note that the gcc extensions makes it possible to check for errors. It is very easy to mix-up entries, or to forget to update the code if the order of the fields change in the structure definition.

This can be written, with the gcc extension as:

```
struct foo x = {
   ...
   .max_rretries = -1,
   .max_wretries = -1,
   .densities = {0x44, 0x44, 0x46, 0x46}
   ...
};
```

This can be written also as a series of affectations, but this force the programmer to repeat the name of the variable each time which is tedious. Note that PL like Pascal or OCaml provide the 'with' feature that avoids this problem. Also for C, the statement affections can not be used at the toplevel, for instance to set global static variables.

Those kinds of comments appears mainly in OpenSolaris. Both Linux and FreeBSD use extensively the gcc designator extension. The question is why OpenSolaris does not use this feature ? Maybe OpenSolaris programmers wanted to make their code more portable and independent of gcc (maybe the Sun or Intel compiler do not support such a feature). In that case it is maybe better to provide, instead of a special feature, a comment annotation that can be used by external checkers. Another solution would be to extract from the gcc compiler this feature and make it independent of gcc and plug-gable into different compilers. This may be the approach advocated by extensible compilers like Xoc [1].

7.7.2 DesignatorMethod()

A specific use of field designators is to mimic object-oriented classes in C by using structures with different function pointer fields to represent the different method of a class.

```
  struct devmap_callback_ctl agp_devmap_cb = {
!!        DEVMAP_OPS_REV,           /* rev */
          agp_devmap_map,           /* map */
          NULL,                     /* access */
          agp_devmap_dup,           /* dup */
          agp_devmap_unmap,         /* unmap */
  };
```

```
struct cpu_functions arm10_cpufuncs = {
          ...
          /* CPU functions */
          cpufunc_nullop,                   /* flush_prefetchbuf    */
          armv4_drain_writebuf,             /* drain_writebuf       */
          cpufunc_nullop,                   /* flush_brnchtgt_C     */
!!        (void *)cpufunc_nullop,           /* flush_brnchtgt_E     */
```

```
!!        nodev,                    /* cb_ioctl */
          nodev,                    /* cb_devmap */
          nodev,                    /* cb_mmap */
          nodev,                    /* cb_segmap */
          nochpoll,                 /* cb_chpoll */
          ddi_prop_op,              /* cb_prop_op */
```

```
          NULL,                             /* bus_dma_ctl */
          tphci_ctl,                        /* bus_ctl */
!!        ddi_bus_prop_op,                  /* bus_prop_op */
          NULL,                             /* bus_get_eventcookie */
          NULL,                             /* bus_add_eventcall */
```

```
          CB_REV,                           /* rev */
          nodev,                            /* int (*cb_aread)() */
!!        nodev                             /* int (*cb_awrite)() */
  };
```

This one is also an example of repeat type.

This can be written, with the gcc extension as:

```
struct cpu_functions arm10_cpufuncs = {
        ...
        /* CPU functions */
        .flush_prefetchbuf = cpufunc_nullop,
        .drain_writebuf = armv4_drain_writebuf,
        .flush_brnchtgt_E = cpufunc_nullop,
        ...
}
```

This can of course also be provided by another PL feature, object-oriented class. In C++ one does not even need to use designators for this. One can simply inherit from a super-class and override or not, by using the same method name, the different methods like this:

```
class arm10_cpufunc : cpu_functions {
 void public flush_prefetchbuf(...) { ... }
 void public drain_writebuf(...) {
   // code of armv4_drain_writebuf directly, which avoid
   // to introduce an extra name like armv4_drain_writebuf
 }
 ...
}
```

So maybe comments led to the invention of designators, which later led to the invention of Object-oriented (but I doubt that).

7.7.3 DesignatorArray()

Designators can also be used with arrays, but are less useful.

```
  unsigned short snd_gf1_atten_table[SNDRV_GF1_ATTEN_TABLE_SIZE] = {
!!   4095 /* 0   */,1789 /* 1   */,1533 /* 2   */,1383 /* 3   */,1277 /* 4   */,
     1195 /* 5   */,1127 /* 6   */,1070 /* 7   */,1021 /* 8   */,978  /* 9   */,
```

```
        0xc1b40000,             /* [147] -22.5 dB */
!!      0xc1b00000,             /* [148] -22.0 dB */
        0xc1ac0000,             /* [149] -21.5 dB */
        0xc1a80000,             /* [150] -21.0 dB */
```

```
        { 0, (sy_call_t *)linux_removexattr, AUE_NULL, NULL, 0, 0 },    /* 235 = linux_rem
!!      { 0, (sy_call_t *)linux_lremovexattr, AUE_NULL, NULL, 0, 0 },   /* 236 = linux_lren
        { 0, (sy_call_t *)linux_fremovexattr, AUE_NULL, NULL, 0, 0 },   /* 237 = linux_fren
        { AS(linux_tkill_args), (sy_call_t *)linux_tkill, AUE_NULL, NULL, 0, 0 },
/* 238 = linux_tkill */
```

```
        { 0, 0, 0, RW_READER, 0 },                                      /* 0x5C 092 */
        { 0, 0, 0, RW_READER, 0 },                                      /* 0x5D 093 */
!!      { 0, 0, 0, RW_READER, 0 },                                      /* 0x5E 094 */
        { 0, 0, 0, RW_READER, 0 },                                      /* 0x5F 095 */
```

Note the use of both hexadecimals and numerals for the index.

```
!!      { 0xaa, 0xaa, 0xaa },   /*  7   2/3  2/3  2/3   White */
          { 0x55, 0x55, 0x55 }, /*  8   1/3  1/3  1/3   Gray  */
        { 0x55, 0x55, 0xff },   /*  9   1/3  1/3   1    Bright Blue  */
```

Here it shows also maybe the need for a real color type (a *Unit* type comment example again).

```
!!              X86_SOCKET_754,         /* 0b00 */
                X86_SOCKET_940,         /* 0b01 */
                X86_SOCKET_754,         /* 0b10 */
                X86_SOCKET_939          /* 0b11 */
        },
```

```
!!      12,                             /* 0x2B: serial */
        0,                              /* 0x2C: timer/counter 0 */
        0,                              /* 0x2D: timer/counter 1 */
        14,                             /* 0x2E: uncorrectable ECC errors */
```

```
        /* 093 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
!!      /* 094 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
        /* 095 */ { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
```

This can be written, with the gcc extension as:

```
int foo[] = {
 ...
 [093] = { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
 [094] = { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
 [095] = { IPI_DONTCARE, 0, 0, 0, NULL, NULL },
 ...
 [100..110] = { IPI_DONTCARE, 0, 1, 0, NULL, NULL },
 ...
}
```

The use of designator arrays is useful when one use range array designator, like in the preceding example for the range 100 to 110.

```
!!      /* 8 */ 0xFE80, 0xFE81, 0xFE82, 0xFE83, 0xFE84, 0xFE85, 0xFE86, 0xFE87,
                0xFE88, 0xFE89, 0xFE8A, 0xFE8B, 0xFE8C, 0xFE8D, 0xFE8E, 0xFE8F,
        /* 9 */ 0xFE90, 0xFE91, 0xFE92, 0xFE93, 0xFE94, 0xFE95, 0xFE96, 0xFE97,
                0xFE98, 0xFE99, 0xFE9A, 0xFE9B, 0xFE9C, 0xFE9D, 0xFE9E, 0xFE9F,
        /* A */ 0xFEA0, 0xFEA1, 0xFEA2, 0xFEA3, 0xFEA4, 0xFEA5, 0xFEA6, 0xFEA7,
                0xFEA8, 0xFEA9, 0xFEAA, 0xFEAB, 0xFEAC, 0xFEAD, 0xFEAE, 0xFEAF,
```

7.7.4 DesignatorHashArray()

```
int foo[] = {
        ...
        NULL,                           /* 0x03, Reserved */
!!      AcpiRsGetVendorLarge,           /* 0x04, ACPI_RESOURCE_NAME_VENDOR_LARGE */
        AcpiRsConvertMemory32,          /* 0x05, ACPI_RESOURCE_NAME_MEMORY32 */
        ...
}
```

```
!!      ISPOPMAP(0x03, 0x03),   /* 0x71: MBOX_FABRIC_LOGOUT */
        ISPOPMAP(0x0f, 0x0f),   /* 0x72: MBOX_INIT_LIP_LOGIN */
        ISPOPMAP(0x00, 0x00),   /* 0x73: */
```

```
  static char *ipsec_policy_failure_msgs[] = {

!!      /* IPSEC_POLICY_NOT_NEEDED */
        "%s: Dropping the datagram because the incoming packet "
        "is %s, but the recipient expects clear; Source %s, "
```

```
        "Destination %s.\n",

        /* IPSEC_POLICY_MISMATCH */
  ...
```

Programmers sometimes (ab)use the designator feature (or comment) to construct hash-tables (in fact to mimic hash-tables) via arrays. As C does not provide any hash-table implementation by default, programmers use the following trick:

This requires first to define the symbolic names that would be the keys to this hash-table:

```
#define IPSEC_POLICY_NOT_NEEDED 0x0
#define IPSEC_POLICY_MISMATCH 0x1
...
```

Then, with this hash-table/array built, the programmer can access from a given key the relevant information:

```
char *message = ipsec_policy_failure_msgs[IPSEC_POLICY_NOT_NEEDED];
```

Note that even if this works (by using comments, or gcc designators), mistakes can be made. Indeed, the use of cpp (again) is error-prone and nothing is really enforced about this special kind of hash-table. For instance, nothing forbid to access those tables directly with integers instead of using the cpp symbolic constants. It would be better to have a direct support for fast hash-tables that does not involve the use of cpp macros.

7.8 ByteRange()

There are lots of such comments but they are almost all in the same few files. This shows again one of the problem in our sampling approach: a file with lots of (possibly auto-generated) comments entail a bias. Maybe the sampling should try to accommodate such case, for example by trying to modulate the sampling by infering the difficulty of the comment, the time it takes to write it. A small comment, or in this case very short comment, with no words at all should take normally less time to write.

Another example of auto-generated comments was described in the *Font* category.

```
static const unsigned char charset2lower[256] = {
  ...
        0x80, 0x81, 0x82, 0x83, 0x84, 0x85, 0x86, 0x87, /* 0x80-0x87 */
!!      0x88, 0x89, 0x8a, 0x8b, 0x8c, 0x8d, 0x8e, 0x8f, /* 0x88-0x8f */
        0x90, 0x91, 0x92, 0x93, 0x94, 0x95, 0x96, 0x97, /* 0x90-0x97 */
        0x98, 0x99, 0x9a, 0x9b, 0x9c, 0x9d, 0x9e, 0x9f, /* 0x98-0x9f */
  ...
}
```

```
        0xCD,0x40,0x09,0x18,0x05,0xFD,0xED,0x2C,        /* 0C50:  @ ...      , */
!!      0xA2,0xFF,0xCD,0x37,0x0C,0x26,0x00,0xC3,        /* 0C58:     7 .&.   */
        0x12,0x0D,0xCD,0x40,0x09,0x26,0x00,0xC3,        /* 0C60: ..  @ .&.   */
```

```
!! /* U+1FDB0 */ IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,   /* U+1FDBF */
   /* U+1FDC0 */ IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,   /* U+1FDCF */
   /* U+1FDD0 */ IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,   /* U+1FDDF */
```

Programmers, or tools, sometimes use a single byte address, or a range, or put the beginning of the range in comment at the beginning of the line and another one for the end of range at the end of the line. Sometimes the address is absolute, and sometimes relative to a constant as in `U0x10+`. The address is usually in hexadecimal.

7.9 ByteAddress()

In big structures, programmers sometimes put the byte address of a field. This may show again that C may not be low-level enough and that programmers want to express properties at the byte level.

```
        uint32_t        micibdbar_reg;          /* 60h - 63h */
        uint8_t         miciciv_reg;            /* 64h - 64h */
!!      uint8_t         micilviv_reg;           /* 65h - 65h */
        uint16_t        micisr_reg;             /* 66h - 67h */
        uint16_t        micipicb_reg;           /* 68h - 69h */
```

```
  typedef struct _txdma_mailbox_t {
        tx_cs_t                 tx_cs;                  /* 8 bytes */
        tx_dma_pre_st_t         tx_dma_pre_st;          /* 8 bytes */
        tx_ring_hdl_t           tx_ring_hdl;            /* 8 bytes */
!!      tx_ring_kick_t          tx_ring_kick;           /* 8 bytes */
        uint32_t                tx_rng_err_logh;        /* 4 bytes */
```

OS code contains many comments about the precise byte layout of large structures such as the one above. Such layouts about devices, network protocols, file systems, etc. are specified in external documents. Such a structure has to follow a predefined layout, e.g., byte `65h` must be the "Microphone In Last Valid Index Value" (MICILVIV) register. Programmers usually put the specification of the layout in comments, e.g., `/* 65h-65h */`, and `/* 66h-67h */`. To follow the specification, programmers have to compute the exact number of bytes and use the right integer type to declare the storage for each field, e.g., `66h-67h` is 2 bytes therefore it should use type `uint16_t`. Such calculation can be Given such specifications, it is tedious and error-prone to write the structure defition when programmers need to handle lots of such structures, each with hundreds of bytes. If the programmer makes a mistake in the calculation, the program can read or write values to the incorrect field and mess up the layout, introducing bugs.

If we design an annotation tag to allow programmers to mark such important byte addresses, e.g., `/* @@byteaddr 66h-67h */`, then we can compare if the code follows the layout specification easily and automatically. Such annotations can make the bug detection process easier and more accurate. In addition, as it is error-prone and inconvenient for developers to calculate how many bytes each field should be, it may be useful to design a domain specific language to semi-automatically generate the structure based on the layout specification.

As typedef are used for most of the fields, those sizes are not so easy to know. Programmers can not use the sizeof() feature of gcc as they want to get the value at compile-time and print it in the file. Maybe the need, again, for some advanced compile-time reflexion capability, is important.

```
struct ohci_registers {
        fwohcireg_t     ver;            /* Version No. 0x0 */
        fwohcireg_t     guid;           /* GUID_ROM No. 0x4 */
        fwohcireg_t     retry;          /* AT retries 0x8 */
#define FWOHCI_RETRY    0x8
        fwohcireg_t     csr_data;       /* CSR data    0xc */
        ...
#define FWOHCI_GUID_H   0x24
#define FWOHCI_GUID_L   0x28
```

```
        fwohcireg_t     guid_hi;        /* GUID hi 0x24 */
        fwohcireg_t     guid_lo;        /* GUID lo 0x28 */
        ...
        fwohcireg_t     dummy0[2];      /* dummy 0x2c-0x30 */
        fwohcireg_t     config_rom;     /* config ROM map 0x34 */
        ...
        fwohcireg_t     hcc_cntl_clr;   /* HCC control clr 0x54 */
#define OHCI_HCC_BIBIV  (1 << 31)       /* BIBimage Valid */
#define OHCI_HCC_BIGEND (1 << 30)       /* noByteSwapData */
#define OHCI_HCC_PRPHY  (1 << 23)       /* programPhyEnable */
        ...
!!      fwohcireg_t     it_int_clear;   /*      0x94 */
        fwohcireg_t     it_int_mask;    /*      0x98 */
```

Note in the preceding comments the use of macros to access some entries by address rather than field names, the use of dummy variables to pad bytes, the use of bitsets here put next to the corresponding variable (a kind of code correlation), and the notion of low and high bytes.

```
        u32     MacRxState;     /* 0x220 */

        u32     pad10[7];

!!      u32     CpuBCtrl;       /* 0x240 */
        u32     PcB;

        u32     pad11[3];

        u32     SramBAddr;      /* 0x254 */
```

Here the address is put at regular intervals, not for each field.

How to check that the comment is right ? Again it would require some reflexion capability over the structure of the program itself.

C can be used to conveniently mmap disk data-structure in memory, and in that case there is a precise disk format with bytes at specific place. Those byte address comments serve such a purpose. For instance in JFS (a filesystem) one can find such code:

```
/*
 * The journal superblock.  All fields are in big-endian byte order.
 */
typedef struct journal_superblock_s
{
/* 0x0000 */
journal_header_t s_header;

/* 0x000C */
/* Static information describing the journal */
__be32 s_blocksize; /* journal device blocksize */
__be32 s_maxlen; /* total blocks in journal file */
__be32 s_first; /* first block of log information */
```

An extension of gcc (yet another one) called 'offsetof' allows to know at compile-time the relative byte address of fields which may make obsolete the previous comments.

```
struct foo {
```

```
 int x;
 float y;
 int :4 z1; //bitfield
 int :12 z2;
 double w;
};
...
  int address = offsetof(foo, y); // should be 4 on as int x takes 4 bytes
```

7.10 Crossref()

```
        if (priv->wep_is_on) {
!!              /* There's a comment in the Atmel code to the effect that this
                   is only valid when still using WEP, it may need to be set to
                   something to use WPA */
                memset(mib.key_RSC, 0, sizeof(mib.key_RSC));
```

```
  /*
   * The cast to int32_t does not result in any loss of information because
   * the number of logical blocks in the file system is limited to
   * what fits in an int32_t anyway.
   */

  #define       lblkno(fs, loc)         /* calculates (loc / fs->fs_bsize) */ \
        ((int32_t)((loc) >> (fs)->fs_bshift))
!! /*
   * The same argument as above applies here.
   */
!!#define       numfrags(fs, loc)       /* calculates (loc / fs->fs_fsize) */ \
```

```
        /* See "auto" comment in init_setup */
        for (i = 1; i < MAX_INIT_ARGS; i++)
```

```
/* see below for values of (this variable) */
```

```
/* see comment in struct sock definition to understand why we need
 * sk_prot_creator
 */
```

7.11 Clone()

Programmers often copy-paste code but feels the need to describe from where the copy, the clone, come from.

```
/*
 * FKS: This is a one-on-one copy of sbpro_audio_set_channels
 * (*) Modified it!!
```

```
 */
static short ess_audio_set_channels(int dev, short channels)
{
        sb_devc *devc = audio_devs[dev]->devc;
```

```
                pci_write_config_dword(de->pdev, PCIPM, pmctl);

!!              /* de4x5.c delays, so we do too */
                msleep(10);
```

```
 /*      Similar to remap_pfn_range() (see mm/memory.c) ... */
```

```
 /* derived from mm/shmem.c and fs/ramfs/inode.c ... */
```

```
!!/*
   * The TCP normal data output path.
   * NOTE: the logic of the fast path is duplicated from this function.
   */
  static void
  tcp_wput_data(tcp_t *tcp, mblk_t *mp, boolean_t urgent)
  {
```

Note that this comment also express a performance semantic property that a profiler could check (see the *Time and Space properties* category).

There are lots of tools for clone-detection like CP-miner [] that try to detect clones, blindly. But they could also use the semantic information provided by the programmer which would make it far easier. Also, after each change on a function clearly annotated with a CloneCode comment, it would be very fast to detect his clones and warn the user that he should also maybe modify the other code in the "clone group". We could have a copy-paste oriented programming paradigm where copy-pasting would not be anymore a problem but in fact embraced. Those annotations could even be for some parts auto-generated by the IDE who knows when the user copy-paste big chunk of code (yet another example of synergy through annotations between tools, here the IDE, VCS, and CP-miner).

```
  #ifndef NO_DUMMY_DECL
  struct internal_state      {int dummy;}; /* for buggy compilers */
  #endif

  #endif
!!/* --- infutil.h */

  #ifndef NO_DUMMY_DECL
  struct inflate_codes_state {int dummy;}; /* for buggy compilers */
  #endif
```

The comment indicates the end of a copy paste. It is also a *EndOfXXX*. Note that the code is copied from another software (from gzip) and so keeping both version synchronized is even more harder. Some VCS provide functionality to handle subsystems, and branches, but do not support such fine grained requirement. As in open-source one can easily include code from another software, and that software evolves a lot, keeping the version up-to-date and bug-free is harder.

```
/*...
**  This driver has been ported to Linux from the FreeBSD NCR53C8XX driver
**  and is currently maintained by
**
**          Gerard Roudier              <groudier@free.fr>
**
**  Being given that this driver originates from the FreeBSD version, and
**  in order to keep synergy on both, any suggested enhancements and corrections
**  received on Linux are automatically a potential candidate for the FreeBSD
**  version.
**
**  The original driver has been written for 386bsd and FreeBSD by
**          Wolfgang Stanglmeier        <wolf@cologne.de>
**          Stefan Esser                <se@mi.Uni-Koeln.de>
**
**  And has been ported to NetBSD by
**          Charles M. Hannum           <mycroft@gnu.ai.mit.edu>
**
**  NVRAM detection and reading.
**    Copyright (C) 1997 Richard Waltham <dormouse@farsrobt.demon.co.uk>
**
**  Added support for MIPS big endian systems.
**    Carsten Langgaard, carstenl@mips.com
**    Copyright (C) 2000 MIPS Technologies, Inc.  All rights reserved.
**
**  Added support for HP PARISC big endian systems.
**    Copyright (C) 2000 MIPS Technologies, Inc.  All rights reserved.
...

/*************** ORIGINAL CONTENT of ncrreg.h from FreeBSD *****************/
struct ncr_reg {
/*00*/  u8 nc_scntl0;    /* full arb., ena parity, par->ATN  */

/*01*/  u8 nc_scntl1;    /* no reset                         */
        #define   ISCON   0x10  /* connected to scsi     */
        #define   CRST    0x08  /* force reset                     */
        #define   IARB    0x02  /* immediate arbitration           */
...
/*
 * End of ncrreg from FreeBSD
 */
```

7.12 Aspect()

Configuration aspect.

```
/*
 *       (DV)     = only defined for Da Vinci
 *       (ML)     = only defined for Monalisa
 */
```

7.13 Misc()

```
        int                     matched;        /* matched the value */
!!      int                     seenzero;       /* saw a 0 bestfree entry */
  #endif
```

```
!!int xdf_fbrewrites;    /* how many times was our flush block rewritten */
```

```
!!/*
   * List of PMC owners with system-wide sampling PMCs.
   */
  static LIST_HEAD(, pmc_owner)                 pmc_ss_owners;
```

```
  #define       ZSRR0_CD                0x08    /* CD input (latched if R15_CD) */
!!#define       ZSRR0_SYNC              0x10    /* SYNC input (latched if R15_SYNC) */
  #define       ZSRR0_CTS               0x20    /* CTS input (latched if R15_CTS) */
  #define       ZSRR0_TXUNDER           0x40    /* (SYNC) Xmitter underran */
```

```
!!      u_char                  used;   /* # slots in use */
        u_char                  avail;  /* where to start scanning */
        u_char                  busy;
```

Chapter 8

Other()

```
/* fprintf(3) macros for unsigned integers. */

  #define       PRIoLEAST64     "llo"   /* uint_least64_t */
!!#define       PRIoFAST8       "o"     /* uint_fast8_t */
  #define       PRIoFAST16      "o"     /* uint_fast16_t */
```

Chapter 9

Discussions

We have seen in previous sections that many comments found an echo in a specific computer science research work.

9.1 C vs other programming languages

It would be interesting to study the comments in modern PL like Csharp or OCaml or Haskell. Some of the OS comments we found have a direct "solution", a direct translation in some features of modern PLs (but not that many maybe as for instance code correlations comments are also needed in OCaml), but the use of those features may also have some limitations leading to different kinds of comments, leading maybe to the invention of new feature for those modern PLs.

9.1.1 Ada

Ada is solution ?

9.1.2 C++

C++ is solution to many of those problems ? - inline - const - constructor/destructor model - template - exn

9.1.3 Java

Java has solutions to some of the problem:
- exception handling

9.1.4 OCaml

OCaml, and other modern functional languages like Haskell, Fsharp, etc are the solution ?

9.1.5 Other

9.2 Proposed major improvements

Our database of examples can be used as a basis for each of the following works, as the starting point.

9.2.1 Migration tool

As a large percentage of comments could be supported if present in a more formal form, in an annotation, then the first tool to build is a migration tool using probably NLP techniques to leverage such comments. By making such comments less fuzzy the barrier of entry for other tools will be lower and tools can then make use of those comments.

On the one hand, there are lots of existing comments containing useful information, on the other hand there are lots of research work with lots of annotation based tools, advanced type systems, and even advanced programming languages, but annotations are not that used, same for advanced type system, and there is only a few mainstream programming languages that succeeded. Those research work ideas were rarely tested, rarely backup by a user study and so it was difficult to see if they are really useful. They had very few impacts on OS code. In fact, maybe the domain where they have the least impact is OS code which is unfortunately arguably the most critical part that should be make more reliable. OS programmers still use C. Maybe because of latency, but maybe there is more.

The works that really succeeded followed an evolutionary approach. C is extended from time to time (lots of gcc extensions). C++ is all about extending C while maintaining backward compatibility. So, even if lots of the previous research works on advanced type system or annotations languages are good, they may have forget maybe one of the most important thing to succeed: provide a migration path that leverage the existing information. Comments can be the basis for such migration.

I have started myself using tags in my own program :)

9.2.2 Unifying framework and generic frontend

Many tools, like Emacs, CVS, Eclipse already use special comments annotations in addition to all the type-based annotation checkers. With so many annotations and tools, we need a unifying framework to make it easier for the programmer to consistently use all those annotations. We need a generic annotation language, possibly with a generic frontend where new checkers and annotations can be plugged-in. It would be useful for instance to be able to use at the same time, as-is, SAL and sparse, or splint and deputy. Each of those annotation languages have their own names for annotations as well as their own specification to place those annotations in the code (before the statement, after, via a comment, via a macro). By having a generic front-end one could easily feed the annotations to the different tools (on Linux using sparse and windows using SAL), and so benefit from all those tools, and especially their internal static analysis algorithms, easily.

9.2.3 Extensible checker

9.2.4 CAP: Computer Assisted Programming

9.2.5 cpp-lint

See Section [?].

9.2.6 Source code visualizer and browser

There are lots of code relationships information that could be leveraged by a tool. An annotation-based guided visualizer and navigation tool could be helpful to help programmers understand and maintain code. I do not speak about hypertext capabilities; there are lots of tools that can already do this I think. I speak about a better source code visualizer that can make the most use of the provided pixels on the screen to display as much contextual information as possibly based on the information in the annotations to enable some focus+context.

9.2.7 NewC

9.2.8 COP: Copy-paste Oriented Programming

9.2.9 Relationship

9.2.10 Anti-devil

9.2.11 Semantic VCS

9.3 Caveats

A typical problem of sampling from a large set of data is that infrequently appearing comments may rarely show up, if show up at all, in the sample and are therefore not studied. Some of the concerns reflected by these comments are important but are just not commonly documented in comments. Although we don't know the exact reason for the scarcity of comments for such concerns, we try to discuss one of them here.

9.3.1 User/Kernel Space

User provided data, especially strings, are generally considered untrusted. Thus, they are not allowed to be passed into certain functions to in order to protect the kernel. We almost saw no comments about user/kernel related concerns. The examples shown below are from comments close to our samples.

```
freebsd/gnu/fs/xfs/xfs_rtalloc.h:154:0

/*
 * Grow the realtime area of the filesystem. */

intxfs_growfs_rt(
        struct xfs_mount        *mp,    /* file system mount structure */
!!      xfs_growfs_rt_t         *in);   /* user supplied growfs struct */
```

```
opensolaris/common/inet/nca/nca.h:164:19

/*
 * Serialization queue type (move to strsubr.h (stream.h?) as a general
 * purpose lightweight mechanism for mblk_t serialization ?). */

typedef struct nca_squeue_s {       uint16_t        sq_state;       /* state flags */

        uint16_t        sq_count;       /* message count */
        ....
        kcondvar_t      sq_async;       /* async thread blocks on */
        kmutex_t        sq_lock;        /* lock before using any member */
        clock_t         sq_awaken;      /* time async thread was awakened */
!!        void            *sq_priv;       /* user defined private */
        kt_did_t        sq_ktid;        /* kernel thread id */

} nca_squeue_t;
```

Although Linux's Sparse annotation `__user` can express such concerns in code, and Linux developers commonly do so (backed up by the large amount of annotations we saw in Linux's code), programmers still user comments to express such concerns sometimes as in the examples above. Note that the existence of

Sparse could be a reason that Linux has few such comments (as programmers use such annotations instead of comments), but it does not explain why FreeBSD and OpenSolaris also do not have many such comments.

As a side note, although there are numerous concerns that cannot be expressed by the programming language, programmers do not document all of them in comments. The question, why programmers choose to document some instead of others, itself is an very interesting area (although beyond the scope of this study).

Chapter 10

Conclusion

Here are general conclusions:

- The *Type*, *Interface*, and *Code Relationships* are surely the most interesting categories, at least for bug findings.
- For most of the comments, including the one in *Explanation*, we can find something, a PL, a tool, a research, that is related to the comment, and that shows that the comment is used because of the limitation of something.

Here are conclusions about the numbers:

- *Interface* and *Type* represents 20% and are strongly related to bug finding. It is far more than 1% and so confirm our hope from iComment that many comments have potential for bug detection.
- *Type* is 10%.
- More that 10% of comments could be covered by existing annotation languages, if they could be used together (SAL+meca+splint+sparse+etc).
- Locking, which is spread in different categories (*Context* and *CodeCorrelation*), represents 5% of the comments.
- *Explanation* is big, 50%, which confirms that people still use comment for "explanation", but at the same time shows that it's not the full story. Many comments are not fuzzy explanation about the code, and may be supported by tools.
- 40% (600 000 comments) could be used to improve software development and maybe reliability.
- even 1% is big as it may represent 14 000 comments
- The *Explanation* number is a lower bound. We classified many comments as explanation maybe because we didn't know or imagined that an existing research work could leverage this comment. For instance if one has never heard about unit and dimensions type system (millisecond, speed, etc), then it's easy to classify such comments as explanation. So when we looked at comments, we had to be very open-minded about the potentiality of the comment, which is very hard and so we may have missed some opportunities. We don't know about all research work done in programming languages or software engineering.
- There is very few really stupid comments that just paraphrase the code like i++; /* increment i */

- There is a number of comments that are used because of really bad coding. For instance the use of magic numbers should be translated at least in symbolic constant definition. In many cases the comment translates to a limitation. So, the more the comments, the worse is the program or the language, which goes against most criteria of software quality which argue for the use of abundant comments.
- Most comments are short. There are very few design comments that explain at length how the code works.
- Most comments are about code properties or organization, and not so much about Linux itself or how an OS work.

Bibliography

[1] Cox, R., Bergan, T., Clements, A., Kaashoek, F., and Kohler, E. Xoc, an extension-oriented compiler for systems programming. In *ASPLOS* (2008).