LCLint: A Tool for Using Specifications to Check Code

David Evans, John Guttag, James Horning, and Yang Meng Tan*

Abstract

This paper describes LCLint, an efficient and flexible tool that accepts as input programs (written in ANSI C) and various levels of formal specification. Using this information, LCLint reports inconsistencies between a program and its specification. We also describe our experience using LCLint to help understand, document, and re-engineer legacy code.

Keywords: C, Larch, LCLint, lint, specifications, static checking

1 Introduction

Software engineers have long understood that static analysis of program texts can both reduce the number of residual errors and improve the maintainability of programs. Traditional static checkers [10, 20] detect type errors and simple anomalies such as obviously uninitialized variables. These checkers demand little effort from the user, and are frequently used. However, their utility is limited by their lack of information about the intent of the programmer. At the other extreme are program verification systems [2]. They are used to demonstrate that a program implements a specification. They require considerable effort from the user, and are seldom used in software development.

Somewhere between these extremes are tools that use formal specifications, but don't attempt complete verification. They are intended to retain the ease of use and efficiency of traditional static checkers while providing stronger checking using the specifications.

The goal of the work presented here is to gain a better understanding of how to build static checkers that allow users to conveniently manage the tradeoff between ease of specification and comprehensiveness of checking and how such tools can aid software development and maintenance. To do this, we built and used a flexible tool, LCLint, that supports different levels of specification and different styles of programming.

The next section outlines the design goals of LCLint and discusses the kinds of checks it performs. Section 3 uses a tiny example to illustrate how LCLint can be used to understand and improve code. Section 4 reports on our experience using LCLint on larger programs. Section 5 discusses related work. Section 6 summarizes what we learned by building and using LCLint. Appendix A provides a comprehensive list of LCLint flags. Appendix B shows how stylized comments can be used for local control of checking. Appendix C describes how to obtain LCLint by anonymous ftp.

2 An Overview of LCLint

LCLint accepts as input programs written in ANSI C and various amounts of specification written in the LCL language [8, 19]. It is intended to be useful in developing new code and in helping to understand, document, and re-engineer legacy code.

Some of our important design goals were:

- Efficiency—Since LCLint should be run whenever the source code or specification is changed, the time needed to run LCLint should be comparable to that for compilation. This limits the checking to simple checks that do not require global analysis.
- Flexibility—LCLint is not intended to impose a specific style of coding. Hence, its checking must be customizable to support different programming styles.
- Incremental effort and gain—LCLint should provide significant benefits without programmers expending much effort on writing specifications. Benefits should increase as further effort is put into the specifications.

* James Horning can be reached at the DEC Systems Research Center, horning@src.dec.com. The other authors are at the MIT Laboratory for Computer Science, [evs,guttag,ymtan]@lcs.mit.edu and are supported in part by ARPA (N00014-92-J-1795), NSF (9115797-CCR), and DEC ERP.

- Ease of use—LCLint should be as easy to run as a compiler, and its output should be easy to understand.
- Ease of learning—LCLint is intended to be an entry into writing formal specifications for programmers who would not otherwise write them, so the knowledge of formal specifications needed to start realizing the benefits of LCLint should be minimal.

While LCLint may be run on any ANSI C program, it cannot do better checking than a traditional lint unless the program conforms to stylistic guidelines or the programmer supplies additional information in the form of partial specifications.

LCLint warns about the following problems, using information not available to traditional static checkers:

- Violation of abstraction boundaries.
  - Failure to properly distinguish between private and public functions, variables, and types.
  - Direct access to the representation of an abstract type in client code.
  - Client code whose meaning might change if the representation of an abstract type were changed.
  - Inappropriate use of a type cast.
  - Exposure of an abstract representation (e.g., client code may gain access to a pointer into an abstract data structure.)
- Undocumented use of global variables.
- Undocumented modification of state visible to clients (through global variables or reference formal parameters.)
- Missing initialization for an actual parameter or use of an uninitialized formal parameter.

The checks that LCLint currently does represent only a fraction of the checking that such a tool could do. However, as indicated in Section 4, even these basic checks offer significant benefits.

LCLint has several checking modes for coarse control of checking, as well as many command line flags to enable and disable specific checks. Regions of code can be annotated to suppress warnings that the user does not wish to see. (See Appendices A and B for details.)

3 The Incremental Use of LCLint

In this section, we interleave a discussion of the kinds of checking done by LCLint with a running example. We show how successively more checking can be performed as style guidelines are adopted or specifications are added. We start with a program that has no specification, then separate interface and implementation information, then introduce an abstract type, then add information about global variables, then say which variables each function may modify, and finally indicate which pointer parameters are to be used only for returning values.

3.1 Checking Raw Code

We begin by looking at the way LCLint's type system and fine-grained control flags can be used to understand the conventions used in legacy code. We start with a module, `date.c` (Figure 1), taken from a C program. We have seeded some errors in the module to illustrate the kinds of errors LCLint can catch. We assume initially that the programmer has not distinguished the types `int`, `char`, and `bool`. (Though C does not have a separate type for the result of logical tests, LCLint treats `bool` as a built-in type.[1])

Running LCLint with the command line

```
lclint +boolint +charint date.c
```

yields no warnings.[2]

We now begin to try to understand the conventions used in the program by running LCLint with various flags. We first test whether or not the program distinguishes `int`, `char`, and `bool` by running LCLint with the command line

```
lclint date.c
```

This generates six warnings:

```
date.c:27,19: Conditional predicate not bool,
                 type int: local
date.c:32,20: Function setToday expects arg 2
                 to be int gets bool: TRUE
date.c:34,20: Function setToday expects arg 2
                 to be int gets bool: FALSE
date.c:60,12: Return value type bool does not
       match declared type int: FALSE
date.c:67,14: Return value type bool does not
       match declared type int: FALSE
date.c:69,14: Return value type bool does not
    match declared type int: ((d1.normal.year
    < d2.normal.year) || ((d1.normal.year ==
    d2.normal.year) && (day_of_year(d1) <
    day_of_year(d2))))
```

The first message is generated because LCLint expects the test of a conditional expression to be a `bool`. The next four messages are generated because the constants `TRUE` and `FALSE` are of type `bool`, but are used where `int`s are expected. The final message is generated because LCLint treats the result of a comparison operator as type `bool`. These messages confirm our hypothesis that the programmer has not distinguished `bool` from `int`. However, it appears that the programmer has not relied upon `char` and `int` being the same.

We now convert to a style of programming that treats `bool` as a distinct type. This involves some editing, replacing `int` by `bool` wherever we intend a logical value. So, for example, the function prototype for the function `date_isInYear` becomes

```
bool date_isInYear (date d, int year);
```

Running LCLint on the revised program yields one warning:

```
date.c:61,10: Return value type int does not
    match declared type bool: (d.normal.year =
    year)
```

[1] A client of the `bool` type includes a standard header defining the `bool` type as `int`, and exporting two constants: `TRUE` and `FALSE`.

[2] The flags `+boolint` and `+charint` indicate that `bool`s, `int`s, and `char`s are to be treated as equivalent.

```
#include <stdio.h>
#include <time.h>
#include "date.h"
#include "error.h" /* defines error */

date today;
date todayGMT;
static int date_tab[13] =
   { 0, 31, 28, 31, 30, 31, 30,
        31, 31, 30, 31, 30, 31 };

#define isLeap(y) \
  ((((y) % 4 == 0) && (((y) % 100) != 0)) \
   || (((y) % 400) == 0))
#define isLeapMonth(m) ((m) == 2)
#define days_in_month(y,m) \
  (date_tab[m] + \
   ((isLeapMonth(m) && isLeap(y)) ? 1 : 0))

int days_in_year (int y)
{ return (isLeap(y) ? 366 : 365); }

void setToday (date * d, int local) {
  char asciDate[10];
  time_t tm = time((time_t *) NULL);
  (void) strftime(asciDate, 10, "%D\0",
27                  local ? localtime(&tm)
                    : gmtime(&tm));
  (void) date_parse(asciDate, d);
}
void setTodayLocal (void)
32 { setToday(&today, TRUE); }
void setTodayGMT (void)
34 { setToday(&today, FALSE); }
void copyDate (date *d1, date *d2) {
36  if (date_isNormal(*d1)) d2->normal = d1->normal;
37  d2->tag = d1->tag;
}
int date_year (date d) {
  if (!date_isNormal(d)) {
    error("year_date expects normal date");
    return -1; }
  return d.normal.year;
}
int day_of_year (date nd) {
  if (!date_isNormal(nd)) {
    error("day_of_year expects normal date");
    return 0;
  } else {
    ndate d = nd.normal;
    int m, day = d.day;
    for (m = 1; m < d.month; m++)
      day += days_in_month(d.year, m);
    return day;
  }
}
int date_isInYear (date d, int year) {
  if (!date_isNormal(d)) {
    error("dateisInYear expects normal date");
60  return FALSE; }
61 return (d.normal.year = year);
}
int date_isBefore (date d1, date d2) {
  if (!(date_isNormal(d1) && date_isNormal(d2)))
    {
      error("date_isBefore expects normal dates");
67    return FALSE;
    } else {
69    return ((d1.normal.year < d2.normal.year)
           || ((d1.normal.year == d2.normal.year)
        && (day_of_year(d1)
            < day_of_year(d2))));
  }
}
```

(`date_parse` removed to save space)

Figure 1: date.c

```
#ifndef DATE_H
#define DATE_H

#include "bool.h"

typedef enum { UNDEFINED, NORMAL } dateKind;
typedef struct {
  int month, day, year; } ndate;
typedef struct {
  dateKind tag; ndate normal; } date;

extern date today;
extern date todayGMT;

extern void setToday (date *d, int local);
extern void setTodayLocal (void);
extern void setTodayGMT (void);
extern void copyDate (date *d1,  date *d2);
extern int date_year (date d);
extern int day_of_year (date d);
extern int days_in_year (int year);
extern int date_isBefore (date d1, date d2);
extern int date_isInYear (date d, int year);
extern int date_parse (char dateStr[],
       date *ind);
extern int date_isNormal (date d);

#define date_isNormal(d) ((d).tag == NORMAL)
#endif
```

Figure 2: date.h

Examining the code, we discover that the implementation of `date_isInYear` returns an `int` because the assignment operator (=) was used where == was intended. We correct this bug, and proceed to the next level, where adding specifications allows additional checking to be done.

3.2 Separating Interfaces from Implementations

A common style for organizing a C program is as a set of program units, often called *modules*. A module consists of an *interface* and an *implementation*. The interface is a collection of types, functions, variables, and constants for use by other modules, called its *clients*. An interface specification provides information needed to write clients.

A C module `M` is typically represented by two files: `M.h` contains a description of its interface, plus parts of its implementation; `M.c` contains most of its implementation, including function definitions and private data declarations. When using LCLint, the role of `M.c` is unchanged, but we move some information previously contained in `M.h` into a separate file, `M.lcl`:

- `M.lcl` contains an interface specification—a formal description of the types, functions, variables, and constants provided for clients as well as comments providing informal documentation. It replaces `M.h` as documentation for client programmers, who should no longer look at `M.h`. The information provided in the specification is also exploited by LCLint to perform more extensive checking than could be done by a traditional lint.

```
typedef enum { UNDEFINED, NORMAL } dateKind;
typedef struct {
  int month, day, year; } ndate;
typedef struct {
  dateKind tag; ndate normal; } date;

date today;
date todayGMT;

void setToday (date *d, bool local);
void setTodayLocal (void);
void setTodayGMT (void);
void copyDate (date *d1,  date *d2);
int date_year (date d);
int day_of_year (date d);
int days_in_year (int year);
bool date_isBefore (date d1, date d2);
bool date_isInYear (date d, int year);
int date_parse (char dateStr[], date *ind);
bool date_isNormal (date d);
```

Figure 3: Client information moved to date.lcl

```
#ifndef DATE_H
#define DATE_H
#include "date.lh"
#define date_isNormal(d) ((d).tag == NORMAL)
#endif
```

Figure 4: New date.h

- `M.h` contains the information needed by the compiler to compile `M.c` and clients of `M`.

LCLint generates a header file, `M.lh`, from `M.lcl` for inclusion in `M.h`. This file contains prototypes for functions and declarations of types, variables and constants specified in `M.lcl`. Automatic generation of `.lh` files saves the user from repeating information from the `.lcl` file in the `.h` file, avoiding an opportunity for error.

Here, we construct `date.lcl` by moving the global types, constants, variables, and prototypes that constitute the interface of the `date` module from `date.h` to `date.lcl`. The files `date.lcl` and `date.h` are shown in Figures 3 and 4.

We now run LCLint with the argument `date`, so that both `date.lcl` and `date.c` are checked. No inconsistencies are reported. This is hardly surprising, since we have merely redistributed information in a more modular way.

3.3 Making a Type Abstract

LCLint categorizes types as exposed or abstract:

- An *exposed type* is a data structure that is described as a C type (e.g., `char *`) that is made known to clients, who are entitled to rely on it.
- An *abstract type* hides representation information from clients, but provides functions to operate on its values. Abstract types are best thought of as collections of related operations on collections of related values [11, 14].

```
   #include <stdio.h>
   #include "date.h"
   #include "error.h"

   int days_between (date startD, date endD) {
6    if (startD.tag != NORMAL
7        || endD.tag != NORMAL) {
       error("days_between expects normal dates");
       return -1;
     }   else if (date_isBefore(endD, startD)) {
     . . .
```

Figure 5: test.c, a trivial client for date

Exposed types correspond exactly to types in C; abstract types do not correspond to anything in C. All the types in our example thus far have been exposed.

C provides no direct support for abstract types, but there is a style of C programming in which they play a prominent role. The programmer relies on conventions to ensure that the implementation of an abstract type can be changed without affecting the correctness of clients. The key restriction is that clients never directly access the representation of an abstract value. All access is through the functions provided by its interface.

An exposed type is specified (in an `.lcl` file) by a C `typedef`. An abstract type is specified by a type declaration and a collection of functions that manipulate values of its type. The representation of these values is hidden within the implementation (in the `.h` file). Clients can create, modify and examine abstract values by calling the functions in the interface. Type checking for abstract types is done by name, except within their implementations, where the abstract type and its representation are equivalent. This allows an implementation to access internal structure that is hidden from clients.

Both *mutable* and *immutable* abstract types are supported. The value of an instance of an immutable type is unchangeable. The value of an instance of a mutable type depends on the computation state. LCLint checking is the same for mutable and immutable types, except for additional checks on mutable types involving modifications and checking that the representation of a mutable type conforms to assignment sharing.

We continue our example by making `date` an immutable abstract data type. To do this, we move the typedefs in `date.lcl` back to the implementation (just before the `#include` of `date.lh`) and replace them in `date.lcl` by the line

```
immutable type date;
```

This set of changes has no effect on checking `date`. It does affect the checking done on clients of `date.lcl`, which may access the structure of dates only through the functions provided in the interface. To see the effect of this checking, we check a sample client module, `test.c`, shown in Figure 5.

Running LCLint with the command line

```
lclint date test.c
```

results in the warnings:

```
test.c:6,7: Access field of abstract type
               (date): startD.tag
test.c:7,10: Access field of abstract type
                (date): endD.tag
```

indicating the two places where `test.c` violates the abstraction boundary of `date`. To repair this, we replace the implementation-dependent expression,

```
startD.tag != NORMAL
```

with the abstract call

```
!date_isNormal(startD)
```

(and likewise for `endD`).

3.4 Function Specifications: Globals

An LCL function specification starts with a *header*, which is a C prototype, extended with a list of the global variables that may be referenced by the function's implementation. It is completed by a *body* enclosed in curly braces. The body contains an optional *requires clause*, an optional *modifies clause*, and an optional *ensures clause*.

Continuing our example, we add globals lists and empty bodies to the function prototypes in `date.lcl`, e.g.,

```
void setTodayLocal (void) date today;
{ }

void setTodayGMT (void) date todayGMT;
{ }
```

We now use LCLint to check that the function bodies reference exactly the intended variables. Running LCLint on the implementation in Figure 1 with the command line[3]

```
lclint -modifies date test.c
```

results in the warnings

```
date.c:34,13: Unauthorized use of global today
date.lcl:10,1: Global todayGMT listed but not
                 used
```

which are both symptoms of using the wrong global variable in the body of `setTodayGMT`.

3.5 Function Specifications: Modifies

A modifies clause says which externally visible objects a function is allowed to change. If there is no modifies clause, then it must not make an externally visible change to the value of any object.

Continuing our example, we add, in `date.lcl`, modifies clauses for the function prototypes of functions that we expect to modify visible objects:

```
void setTodayLocal (void) date today;
{ modifies today; }

void setTodayGMT (void) date todayGMT;
{ modifies todayGMT; }

void setToday (date *d, bool local)
{ modifies *d; }

void copyDate (date *d1, date *d2)
{ modifies *d1; }
```

Running LCLint on the implementation in Figure 1 with the command line

```
lclint date test.c
```

results in the warnings

```
date.c:36,27: Suspect modification of
    d2->normal: d2->normal = d1->normal
date.c:37,3: Suspect modification of d2->tag:
               d2->tag = d1->tag
```

which are both symptoms of assignments in the wrong direction in the body of `copyDate`.

We correct those errors and proceed.

3.6 Use Before Definition

Like many static checkers, LCLint detects instances where the value of a location may be used before it is defined. This analysis is usually done at the function level. If there is a path through the function that uses a local variable before it is defined, a use-before-definition warning is issued.

Detecting use-before-definition errors involving parameters is more difficult. In C, it is common to use a pointer to an undefined value as an argument intended only to receive a return value. Without specifications, we must either do global analysis to detect use-before-definition errors across function calls, or make assumptions that lead to spurious warnings or undetected errors.

In a function specification header, the `out` type qualifier indicates a pointer formal or mutable abstract objects that is meant only to receive a result value. The value pointed to by an `out` parameter is assumed to be undefined when the function is entered. LCLint will generate a warning if it is used before it is defined in the body. All other parameters are assumed to be defined upon entry. LCLint will generate a warning at the point of call if a function is called with an undefined argument, unless it is specified as an `out` parameter.

Continuing our example, we add the `out` type qualifier to three specifications:

```
void setToday (out date *d, bool local)
{ modifies *d; }

void copyDate (out date *d1, date *d2)
{ modifies *d1; }

int date_parse (char dateStr[],
                out date *ind)
{ }
```

Running LCLint on the implementation in Figure 1 now yields the message

```
date.c:36,25: Variable d1 used before set
```

The problem here is that the body of `copyDate` tests the `tag` field of `d1` when it should have tested the `tag` of `d2`. This

[3] The `-modifies` flag is used to suppress messages regarding modification errors. This will be discussed in the next section.

is related to the errors detected by the modifies checking discussed above. The original implementation uniformly reversed `d1` and `d2`.

4 Experience using LCLint

We don't yet have signficant experience using LCLint to develop completely new code. We do, however, have some experience using LCLint to understand and maintain legacy code. We have used LCLint on several programs including:

- A small database program, formally specified in [8].
- `pm`, an 1800-line portfolio management tool that had been in use for several years. This program had no formal specifications, but was structured around abstract data types.
- `quake`, a 5000-line program for automating system builds in Modula-3. We had not looked at this program before running LCLint on it. It had no formal specifications, and we had no idea what style of programming had been employed.
- LCLint itself.

Running LCLint on the database example did not find many significant problems [4]. It did uncover two abstraction violations, and one undocumented modification that revealed a memory leak. It also generated five spurious modification warnings, because of LCLint's imprecise modifies checking. For example, it failed to determine that a series of assignments to an object culminated in restoring the object's initial value.

The `pm` program was used in a study of how formal specifications could facilitate a software re-engineering process aimed at making existing programs easier to maintain and reuse [19]. We wrote LCL specifications for the main modules of the program, and then tried to improve them. Modifications to the program were driven by changes made to the specifications of its modules. Each time the specification of a module changed, the code was revised. We then used LCLint to check the revised code against its new specification. In the process, LCLint uncovered various inconsistencies between the implementations of the modules and their specifications, including abstraction violations, unsanctioned object modifications, and unsanctioned global variable accesses.

The most illuminating experiment was using LCLint on `quake` [4]. We applied it in the manner described in Section 3. We found two bugs (which could also have been found by a conventional lint) by running LCLint on the `quake` source without any specifications. We also learned that the programmer had not distinguished between types `int` and `char`, but almost always treated `int` and `bool` as distinct. Three minor changes were all that was required to make this distinction complete.

Then we used LCLint to discover which types were treated as abstract, by declaring one type at a time to be abstract, and inspecting the messages generated by LCLint. One type was used completely abstractly, and several were so close to being used abstractly that we decided that the abstraction violations were unintended and changed the program to eliminate them. Eliminating these abstraction violations made the client code shorter and simpler. Furthermore, through this process we gained an understanding of the code, which we recorded in the specifications. In particular, the abstract type declarations make clear which types can be safely modified in isolation and supply information about the level of detail at which one needs to read various parts of the program.

Finally, we moved the prototypes from the `.h` files to `.lcl` files. Since we didn't know what the various functions were supposed to do, we did not attempt to construct globals or modifies clauses. We merely ran LCLint with globals and modifies checking turned on. This yielded messages reporting the global variables accessed and the objects modified by each function. We then used these messages to add globals and modifies clauses to the specifications of the functions—thus greatly improving the documentation of the program.

Towards the end of LCLint's development, we began using it on its own code and specifications. Many errors were caught, most involving violations of type abstractions. Most of the actual bugs detected by LCLint were not related to specification checking, although the flexibility and strict type checking provided by LCLint helped us discover errors that we would not have found with a traditional lint. Relatively late in the development of LCLint, we decided that the underlying implementation for representing types was too inefficient. Without LCLint, we would have been reluctant to reimplement such a pervasive type for fear that unexpected dependencies on the previous implementation would lead to bugs that would be hard to find. By using LCLint, however, we could verify that the type was truly abstract, and change its implementation without concern that it might introduce bugs elsewhere.

5 Related Work

Several checkers have been developed to analyze programs using some form of formal specifications.

Cesar [16] allows programmers to specify sequencing constraints for an abstract type using a specification language based on regular expressions. For example, a programmer could specify a file type that may be opened, written to multiple times, and closed, in that order. The prototype Cesar system was too inefficient to be a useful tool in real software development. Cesar built on other systems [7, 20] that use sequencing constraints to find errors in code.

Inscape [17] uses a specification language that can specify pre-conditions and post-conditions of a procedure, as well as obligations on the caller following return from the call (such as closing a returned file). Inscape propagates the specifications of procedures in a program using a special propagation logic incorporating unknown and possible values. Bugs are detected when a pre-condition or an obligation is contradicted. LCL provides no way to express obligations on the caller after the called function returns. Some useful checking could be done if specifications could require, for example, that the caller eventually free a returned object, or that it not modify the returned object.

Aspect [9] is a system for efficiently detecting bugs, by looking for unsatisfied dependencies. The specification language describes dependencies between "aspects" of objects (such as an array's size) in the post-state and pre-state, and the checker reports when a specified dependency is not present in the im-

plementation. Dependency information in LCL specifications is often not available, or is hidden within the specification. Moreover, LCL has no notion of aspects of an abstract type, so it cannot provide the information for some of the checking done by Aspect. Every error reported by Aspect is guaranteed to be an error in the code or the specification. LCLint doesn't provide such a guarantee—some spurious warnings may be generated, but they can all be turned off by the user.

Other tools have been developed more along the lines of improving lint. CCEL [3] is a metalanguage that allows programmers to express constraints that can be checked automatically about C++ programs. Constraints are specified in a language similar to C++, and can constrain design, implementation, and style. Constraints are lexical in nature, but general enough to catch some high-level C++ errors such as flaws in the inheritance hierarchy. CCEL differs from LCLint in that the specifications describe general constraints and naming conventions, but do not specify the properties of specific functions.

LCLint can be viewed as a tool for promoting modular software designs and abstract data types in C. Many modern languages, including C++ [18], Ada [1], Modula-3 [15] and CLU[11], support both. C++ adds support for abstract types and data encapsulation to C, using an object-oriented paradigm. For programmers who need type inheritance, using C with LCLint is not an alternative to C++. But for C programmers who merely wish to use modules and abstract types, LCLint provides data encapsulation and type safety without the overhead and complexity of C++. The other checking done by LCLint is useful in both C and C++.

Like LCLint, the Fortran Abstract Data (FAD) system [13] adds abstract types to an existing programming language. It extends the syntax of Fortran and provides a preprocessor to convert FAD declarations into standard Fortran. Programs using FAD abstract types cannot be compiled by a standard Fortran compiler or readily understood by an experienced Fortran programmer with no knowledge of FAD. In contrast, LCL specifications used by LCLint are orthogonal to the code: the source code is standard ANSI C.

6 Summary and Conclusions

In this paper, we have tried to give a flavor of the kinds of things LCLint can do, but we made no attempt to be comprehensive. A report by Evans [4] contains a complete description of LCLint, and a discussion of how it can be used in developing new code, and in understanding and maintaining legacy code. Tan [19] presents a programming methodology based on the use of LCL and LCLint.

LCLint detects inconsistencies between code and a combination of specifications and programming conventions. Sometimes warnings expose errors in the specifications or in the code. Sometimes they indicate a violation of a programming convention. Such a violation might not be a bug. However a warning that the code depends on implementation details not apparent in the specification, or that it violates conventions upon which other parts of the program may rely, can help programmers produce better programs and better documentation, and can decrease maintenance costs.

LCLint was designed report as many real problems as possible, while generating relatively few spurious warnings. Most checks are sound and complete—it is possible to determine and report exactly those cases where a particular problem is present. Some of the checks involving use-before-definition and modification are imprecise. There are cases where LCLint cannot determine if a suspected problem is present, so a message may be issued for a non-existent problem. In other cases, a real problem may go undetected. Early experience with LCLint [4, 19] suggests that relatively few spurious warnings are generated, and that the available command line options and syntactic comments are adequate to suppress inappropriate messages. It is more difficult to assess the impact of incomplete checks, since we cannot know how many undetected problems exist.

We originally developed LCLint as a tool to detect bugs in programs. We haven't yet had any significant experience using it while developing new programs. (LCLint was well along in development before it was ready to check itself.) Since we mostly applied LCLint to well-tested code, it is not surprising that we mostly got warnings about violations of data abstractions and style conventions, rather than about bugs. But it did expose a few bugs in the well-tested code by reporting inconsistencies with the specifications. Our experience shows that LCLint is also useful in improving code quality, supporting data abstraction, and detecting flaws in specifications.

LCLint does not yet take full advantage of complete LCL specifications. We plan to explore the benefits of more extensive checking. It is not yet clear where we will reach the point of diminishing returns.

Acknowledgments

LCLint is the result of a joint R&D project (Larch) involving Digital and MIT. The other participants in this project, Gary Feldman, Steve Garland, Kevin Jones, Bill McKeeman, Joe Wild and Jeannette Wing, all contributed ideas and/or code that helped immensely. Special thanks are due to Steve Garland who has contributed every step of the way. Also, thanks to Steve Harrison for providing the `quake` example.

References

[1] The Ada programming language reference manual. ANSI/MIL-STD 1815A, US Department of Defense, US Government Printing Office, February 1983.

[2] Dan Craigen. "Verification Environments," *Software Engineer's Reference Book*, edited by John A. McDermid, CRC Press, 1993.

[3] Carolyn K. Duby and Scott Meyers and Steven P. Reiss. "CCEL: A Metalanguage for C++," *USENIX C++ Conference Proceedings*, August 10-13, 1992.

[4] David Evans. *Using Specifications to Check Source Code*, MIT/LCS/TR-628, MIT Laboratory for Computer Science, June 1994.

[5] David Evans. *LCLint User's Guide*, Version 1.4. September 1994. Available in:

```
ftp://larch.lcs.mit.edu/pub/Larch/
   lclint/lclint1.4.userguide.ps.Z
```

[6] G. Feldman and J. Wild. "The DECspec project: tools for Larch/C," *Proc. Fifth Int. Workshop on Computer-Aided Software Engineering*, Montreal, Jul. 1992. Revised version in [12].

[7] L. D. Fosdick and Leon J. Osterweil. "Data flow analysis in software reliability," *ACM Computing Surveys*, 8(3), September 1976.

[8] J.V. Guttag and J.J. Horning with S.J. Garland, K.D. Jones, A. Modet, and J.M. Wing. *Larch: Languages and Tools for Formal Specification*, Springer-Verlag, 1993.

[9] Daniel Jackson. *Aspect: A formal specification language for detecting bugs*, MIT/LCS/TR-543, MIT Laboratory for Computer Science, June 1992.

[10] S.C. Johnson. *Lint, a C Program Checker*, Unix Documentation.

[11] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*, MIT EECS Series, MIT Press, 1986.

[12] U. Martin and J.M. Wing. *Proc. First Intl. Workshop on Larch*, Dedham, Jul. 1992, Springer-Verlag, 1993.

[13] Keith W. Miller, Larry J. Morell, and Fred Stevens. "Adding data abstraction to Fortran software," *IEEE Software*, November 1988.

[14] James H. Morris, Jr. "Types are Not Sets," *First ACM Symp. Principles of Programming Languages*, Boston, Oct. 1973.

[15] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall, 1991.

[16] Kurt M. Olender and Leon J. Osterweil. "Interprocedural static analysis of sequencing constraints," *ACM Transactions on Software Engineering and Methodology*, 1(1), January 1992.

[17] Dewayne E. Perry. "The logic of propagation in the Inscape environment," In *Proceedings of the ACM SIGSOFT'89 Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, 1989.

[18] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[19] Yang Meng Tan. *Formal Specification Techniques for Promoting Software Modularity, Enhancing Software Documentation, and Testing Specifications*, MIT/LCS/TR-619, MIT Laboratory for Computer Science, June 1994.

[20] Cindy Wilson and Leon J. Osterweil. "Omega—a data flow analysis tool for the C programming language." *IEEE Transactions on Software Engineering*, SE-11(9), September 1985.

A Flags

This appendix describes LCLint command line options. It is extracted from the *LCLint User's Guide*, version 1.4 [5].

So that many programming styles can be supported, LCLint provides many flags. Modes are provided for setting many flags at once. Individual message flags override the setting in the mode. Flags listed before the mode have no effect.

Flags are preceded by `+` or `-`. When a flag is preceded by `+` it is *on*; when it is preceded by `-` it is *off*. This convention is clear and concise, but is not standard in UNIX; since it is easy to accidentally use the wrong one, LCLint issues warnings when a user redundantly sets a flag to the value it already had (unless `-warnflags` is used to suppress such warnings). The precise meaning of on and off depends on the particular flag.

Default flag settings are read from `~/.lclintrc` if it is readable. Command-line flags override settings in this default file. The syntax of the `.lclintrc` file is the same as that of command-line flags, except that flags may be on separate lines and the # character may be used to indicate that the remainder of the line is a comment.

Flags can be grouped into four major categories: mode flags for coarse control of LCLint checking, message control flags for selecting specific classes of messages to be reported or suppressed, type equivalence flags for denoting particular types as equivalent or distinct, and general flags for controlling high level behavior. General flags are applicable only at the command line; all other flags may be used both at the command line and in control comments.

Mode Flags

Mode flags set many type equivalence and message control flags to predefined values. A mode flag has the same effect when used with either `+` or `-`. These are brief descriptions to give a general idea of what each mode does. To see the exact flag settings in each mode, use `lclint -help modes`.

`weak` weak checking, intended for typical C code. No modifies checking, macro checking, rep exposure, or clean interface checking is done. Return values of type `int` may be ignored. The types `bool`, `int`, `char` and user-defined `enum` types are all equivalent. Old style C declarations are unreported. Globals checking and predicate checking is done.

`standard` the default mode. All checking done by `weak`, plus modifies checking, global alias checking, use all parameters, ignored return values or any type, macro checking, unreachable code, infinite loops, and fall-through cases. The types `bool`, `int` and `char` are distinct. Old style C declarations are reported.

`checks` moderately strict checking. All checking done by `standard`, plus must modification checking, rep exposure, return alias, and clean interfaces.

`strict` extremely strict checking. All checking done by `checks`, plus modifications and globals used in unspecified functions, unspecified use and modifications of standard streams (`stdio`), and strict typing of C operators.

Message Control Flags

Message control flags are preceded by a `-` to turn the message off, or a `+` to turn the message on. Each flag is described by the class of messages that are reported when it is on, and suppressed when it is off.

- Globals and Modifies Checking

  `+globals` unspecified use of global variable
  `+globunspec` use of global in unspecified function
  `+globuse` global listed for a function not used
  `+modifies` unspecified modification of visible state
  `+mustmod` specified modification is not detected
  `+modunspec` modification in unspecified function
  `+stdio` unspecified use or modification of standard stream (`stdio`, `stdout`, `stderr`)

- Clean Interfaces

  `+specundef` function or variable specified but never defined
  `+export` variable, function or type definition exported but not specified (equivalent to `exportvar`, `exportfcn` and `exporttype`)
  `+exportvar` variable exported but not specified
  `+exportfcn` function exported but not specified
  `+exporttype` type definition exported but not specified

- Declarations

  `+topuse` declaration at top level not used
  `+paramuse` function parameter not used
  `+varuse` variable declared but not used
  `+fcnuse` function declared but not used
  `+overload` library function overloaded
  `+incondefs` function or variable redefined with inconsistent type

- Type Checking

  `+bool` representation of `bool` is exposed
  `+pred` type of condition test (for `if`, `while` or `for`) not boolean
  `+predptr` type of condition test not boolean or pointer
  `+ptrarith` arithmetic involving pointer and integer
  `+ptrcompare` comparison between pointer and number
  `+boolcompare` comparison between `bool`s[4]
  `+strictops` primitive operation does not type check strictly

- Return Values

  `+returnval` return value ignored
  `+returnvalbool` return value of type `bool` ignored
  `+returnvalint` return value of type `int` ignored

- Rep Exposure and Aliasing

  `+repexpose` abstract representation is exposed (equivalent to `retexpose` and `assignexpose`)
  `+retexpose` – abstract representation is exposed (return values only)
  `+assignexpose` abstract representation is exposed (assignments only)
  `+retalias` function returns alias to parameter or global
  `+globalias` function returns with global aliasing external state

- Macros

  `+macroundef` undefined identifier in macro
  `+macroparens` macro parameter used without parentheses
  `+macroparams` macro parameter is not used exactly once
  `+allmacros` check all parameterized macros as functions (macros are not expanded)

- Others

  `+ansi` warn about old style function declarations
  `+infloops` likely infinite loop is detected
  `+casebreak` non-empty case in a switch without preceding break
  `+unreachable` code detected that is never executed

[4] This is dangerous, because there are many possible true values in C. The result of a comparison between two true `bool`s is not necessarily `TRUE`.

Type Equivalence Flags

`+boolint bool` and `int` are equivalent

`+charindex char` can be used to index arrays

`+charint char` and `int` are equivalent

`+enumint enum` and `int` are equivalent

`+forwarddecl` forward struct and union declarations of pointers to abstract representation match the abstract type

`+numliteral int` literals can be floats

`+voidabstract` allow `void *` to match pointers to abstract types

`+zeroptr` literal `0` can be treated as a pointer

General Flags

These flags have the same meaning when used with either `+` or `-`. They control initializations, message printing, and other behavior not related to specific checks.

`help` on-line help. Help can be followed by a topic (available topics shown if no argument is used) or list of flags.

`dump` *file* save state in *file* (default extension `.lldmp`)

`load` *file* load state from *file* (instead of standard library file)

`nolib` do not load standard library

`whichlib` show pathname and creation information for standard library

`i` *file* set LCL initialization file

`f` *file* load options from file

`nof` do not load default options file (`~/.lclintrc`)

`noaccess` ignore access and noaccess comments

`nocomments` ignore all stylized comments

`I`*directory* add *directory* to C include path

`S`*directory* add *directory* to search path for LCL specs

`D`*initializer* define initializer (passed to C preprocessor)

`U`*name* undefine identifier (passed to C preprocessor)

`tmpdir` *dir* set directory for writing temporary files

`showfunc` show name of function containing error (first error in function only)

`singleinclude` optimize include files

`stats` display information on number of lines processed and execution time

`nolh` suppress generation of `.lh` files

`quiet` suppress herald and error count

`expect` *n* set expected number of code errors)

`lclexpect` *n* set expected number of specification errors)

`limit` *n* suppress *n*th and higher consecutive similar messages

`linelen` *n* set length of messages in characters

The following flags have different meanings if `+` or `-` is used. The default behavior is on, described below. Using `-`*flag* has the opposite effect.

`+warnflags` warn when command line sets flag to default value in mode

`+showcolumn` show column number where error is found

`+hints` provide hopefully helpful hints

`+accessunspec` representations of abstract types are accessible in unspecified function in the `.c` file with the same name as the specification

B Control Comments

To provide source level control of LCLint, stylized C comments may be used. All control comments begin with `/*@` and are ended like normal C comments.

Any of the message control and type equivalence flags can be set locally using control comments. At any point in a file, a control comment can set the flags locally to override the command line settings. The original flag settings are restored before processing the next file. The syntax for setting flags in control comments is the same as that of the command line, except that flags may also be preceded by `=` to restore their setting to the original command-line value. For instance,

```
/*@ +boolint -modifies =charint */
```

makes `bool` and `int` indistinguishable types, turns off modifies checking, and restores the equivalence of `char` and `int` to its command line or default setting. This is useful to turn of a particular check for a small segment of code where a convention is consciously violated.

For coarser control, `/*@ignore*/` and `/*@end*/` can be used to suppress all messages. No errors will be reported in code between `/*@ignore*/` and `/*@end*/`. The `ignore` and `end` comments must be matched—a warning is printed if the file ends in an ignore region.

The control comment `/*@i*/` will suppress reporting of any errors from here to the end of the line. The syntax `/*@i`*n*`*/` will suppress any errors from here to the end of the line, but reports a warning message unless exactly *n* errors are found.

Control comments may also be used to override type access settings. `/*@access` *t*`*/` allows succeeding code to access the representation of *t*. To disallow access to abstract type *t*, use `/*@noaccess` *t*`*/`. Both `access` and `noaccess` may be given a list of types separated by spaces. Type access applies from the point of the comment to the end of the file or the next access control comment for that type.

When `+allmacros` is used, `/*@notfunction*/` can be used to indicate that the next macro definition is not intended to be a function, and should be expanded in line instead of checked as a macro function definition. However, `+nocomments` does not cause `/*@notfunction*/` comments to be ignored, because of their syntactic role.

C Availability

LCLint is available via anonymous ftp from

```
ftp://larch.lcs.mit.edu/pub/Larch/lclint
```

Look at `README` in this directory for more information. If you have problems installing LCLint, send a message to

```
lclint@larch.lcs.mit.edu
```

Several platforms are supported, including DEC Alpha AXPs running OSF/1, DECstations running Ultrix, Sun workstations (Sparc) running Solaris 2, Sun workstations (Sparc) running SunOS4.1/Solaris 1, and PCs running linux. Source code is available if you would like to build a version for some other platform.

There are two mailing lists associated with LCLint.

- `lclint-announce@larch.lcs.mit.edu`
  Reserved for announcements of new releases and bug fixes.
- `lclint-interest@larch.lcs.mit.edu`
  Informal discussions on the use and development of lclint.

To subscribe to a list, send a (human-readable) message to

```
lclint-request@larch.lcs.mit.edu
```

The URL for the LCLint home page in the World-Wide Web is

```
http://larch-www.lcs.mit.edu:8001/
larch/lclint.html
```

(without the line break).