CCodeExtractor:一种针对C程序自动化的函数提取方法

张其良 张 昱 周 坤

(中国科学技术大学计算机科学与技术学院 合肥 230026)

摘 要 随着程序规模和复杂性的增加,代码重构在改善软件质量、性能以及提高软件的扩展性和维护性等方面至关重要。目前的Eclipse中,C源代码重构工具的函数提取只能处理一些简单的代码,且处理过程需要人工参与,不能自动化处理。为此,提出一种C源代码级别自动化的函数提取方法(CCodeExtractor),它通过指定提取条件,在保证语义一致的前提下,将符合条件的代码片段自动分离成一个单独的函数,并将其放到新文件中,而原代码片段替换成了一个函数调用。为了验证CCodeExtractor的有效性,结合循环分析和优化在实际应用中的广泛应用,在LLVM上实现了一个提取循环的工具,将程序中满足条件的for循环封装成单独的函数。在实验中,使用该工具对6个不同规模的程序进行了测试并且对比了变换前后程序运行的结果。实验结果表明,CCodeExtractor代码提取方法在保证程序语义不变的前提下,适用于不同规模的程序。

关键词 C程序重构,程序变换,程序理解,代码提取,循环分析,LLVM

中图法分类号 TP311 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2017.04.004

CCodeExtractor:Automatic Approach of Function Extraction for C Programs

ZHANG Qi-liang ZHANG Yu ZHOU Kun

(School of Computer Science and Technology,University of Science and Technology of China,Hefei 230026,China)

Abstract As program complexity increases,code refactoring plays an important role in improving quality and performance of software,and is also essential for improving the maintainability and extensibility of programs. Current code extraction approach for C code in Eclipse,can only deal with some simple code,and cannot refactor the code automatically. In this paper,we proposed a code extraction approach for C code,CCodeExtractor. It can extract the C code fragments that meet the specified conditions into new functions,and replace the fragments with these new functions calls automatically. The refactored code has the same program semantics as the original code. In order to verify the validity of CCodeExtractor,we implemented it in LLVM to extract some for statements in C programs into new functions since loop analysis and optimization has been widely explored in recent years. Our experiments evaluated CCodeExtractor using six actual applications of different size,and compared the outcomes of the original and transformed programs. Experimental results show that CCodeExtractor can provide correct source-level transformation for programs of different size.

Keywords C program refactoring,Program transformation,Program comprehension,Code extraction,Loop analysis,LLVM

1 引言

代码重构[1]是一种改变程序结构但不改变程序行为的处理过程,主要目的是通过调整程序代码来改善软件质量和性能,使程序的设计模式和架构更趋合理,以提高软件的扩展性、维护性。代码变换存在很多形式[2-4],其中函数提取(Extract Method)可以将代码划分为更小、更易理解的程序片段,并将其转化为一个新函数,原程序片段替换为对这个新函数的调用。目前针对C程序的函数提取已经被集成到Visual Studio[5],Eclipse[6]等开发环境中,但它们的提取方法存在一些局限性:1)只能处理一些简单的代码片段(如无法处理包含跳转语句的代码片段);2)在程序变换过程中需要人工参与,不能自动化处理。图1是Eclipse函数提取的一个例子,变换后的程序forfunc函数存在一些错误:1)虽然将multi的地址传到了新函数,但是新函数对multi的使用存在错误,需要人工改成 * multi= * multi * arr[i]; 2)宏N找不到定义。文献[7]虽然在代码重构中考虑了宏定义、条件编译因素,但是其对预处理行为的分析目前并没有应用到C代码方法提取

到稿日期:2015-11-30 返修日期:2016-02-26 本文受国家863高技术研究发展计划项目基金(2012AA010901),国家自然科学基金(61170018)资助。

张其良(1989—),男,硕士生,主要研究方向为程序分析、确定性并行,E-mail:qiliangzh@gmail.com;张 昱(1972—),女,博士,副教授,主要研究方向为面向多核的高效并行系统软件的构建与评估、软件安全等;周 坤(1991—),女,硕士生,主要研究方向为程序分析、自动并行化。

中。除此之外，LLVM[8-9]实现了一个函数提取的模块[10]，在 IR 层次将一段中间代码包装成一个函数调用。由于变换后的程序还是中间代码的形式，源代码层次的一些信息已被处理掉，这给程序员观察和分析程序特点带来了很大的挑战。

```
int main(void) {
  int i,multi=1,sum=0;
#define N 5
  int arr[N]= {1, 2, 3, 4, 5};
  for (i=0;i < N;i++) {
    sum +=arr[i];
    multi=multi * arr[i];
  }
  return sum+multi;
}
```

```
int forfunc(int i,int sum,
    int arr[N],int * multi) {
  for (i=0; i<N; i++) {
    sum+=arr[i];
    multi=multi * arr[i];
  }
  return sum;
}
int main(void) {
  int i,multi=1,sum=0;
#define N 5
  int arr[N]={1,2,3,4,5};
  sum = forfunc (i, sum, arr, &multi);
  return sum + multi;
}
```

图 1　Eclipse 中函数提取的使用示例

针对这些问题，本文提出了一种 C 源代码级自动化的函数提取方法（CCodeExtractor），其通过指定一些条件，在保证语义的前提下，自动将满足条件的代码片段提取出来包装成函数，并放到一个单独的文件中。相比已有的方法，CCodeExtractor 解决了函数提取中的预处理（宏与条件编译）、全局或局部变量引起的参数传递和返回值处理以及跳转语句等问题，能自动提取所有满足条件的代码片段并自动产生保持原程序语义的正确重构的代码。CCodeExtractor 将新函数的定义放到了一个单独的文件中，虽然增加了处理的难度（需要处理公用信息，如头文件、类型声明等），但也带来了一些好处：一方面更便于程序员查看新函数中程序片段的特点，另一方面新函数可以单独被处理（如单独编译、分析或调度等）。CCodeExtractor 除了可以帮助程序员快速地定位代码、查看和理解程序外，还有一些其他潜在的应用：在程序分析领域，CCodeExtractor 可以将程序中的特殊片段（耗时多的 for 循环）从原过程中提取出来形成一个单独的函数，然后单独分析、优化提取后的新函数即可；在程序并行化领域，可以将其中计算密集型的、并行度高的程序片段提取出来包装成函数，再将它们变换成 CUDA 程序放到 GPU 等计算部件执行。

本文有如下贡献：1）提供了一个 C 源代码级别自动化的函数提取方法（CCodeExtractor），不仅便于程序员查看和理解程序的特征，还有助于程序的分析优化工作；2）基于 CCodeExtractor，在 LLVM 上实现了一个循环提取的工具，可以将程序中大于一定迭代次数的循环包装成一个函数调用。实验结果表明：该工具在保证正确变换的前提下，适用于不同规模的代码重构。

本文第 2 节介绍 CCodeExtractor 代码提取的例子；第 3 节介绍 CCodeExtractor 主要的设计和实现策略；第 4 节是实验和分析；最后总结全文。

2　一个 CCodeExtractor 代码提取的例子

本节主要通过图 2(a)介绍 CCodeExtractor 的主要功能以及代码提取过程中的关键问题。图 2(a)中的 example_old.c 是从实际应用程序 SIFT[11]中的函数 isPeak()抽象出来的，这个函数主要用于判断给定点(x,y)是否是给定图片 pic 的顶点。使用 CCodeExtractor 对文件 example_old.c 进行处理后，产生了 3 个文件：图 2(b)中包含了原文件的函数实现，其中一些代码片段被替换成了函数调用；图 2(c)包含了原文件中包含的部分头文件、用户自定义类型、新函数的声明等信息；图 2(d)主要包含了新函数的定义。为了保证变换前后程序语义的一致性，需要考虑收集哪些信息、程序变换的策略以及代码重组的形式。

```
1 #include"..."
2 typedef ... T;   (1)
3 T *g;   (2)
4 bool func(int x, int d) {
5    int i, ... ;
6    switch(d){
7        case 0:  ... ; break;
8        ...;
9        case 3:  ... ; break;
10       defualt: return 0;
11   }
12   for(i = x; ... ; i += inc) {
13       ... = g;   (3)
14       if( cond1 ) return false;
15       if( cond2 ) {
16           ...;  break;}
17   }
18   for(i = x; ... ; i -= inc) {
19       if( cond3 )  goto label;
20       if( cond4 ) {
21           ...;  break;}
22   }
23   label: ...;
24   return ... ;
25 }
```

(a) example_old.c

```
1 #include "example_loop.h"
2 T *g;   (2)
3 bool func(int x, int d) {
4    int i,  ...;
5    switch(d){
6        case 0: ... ; break;
7        ...;
8      case 3: ... ; break;
9      defualt: return 0;
10   }
11   {  int ret1 ; bool ret2;
12      ret1 = func_lp_1(&i, &x,
13 (3)          &inc,&ret2, ...);
14      if(ret1 == 0 ) return ret2;
15   }
16   {  int ret1; bool ret2;
17      ret1 = func_lp_2(&i, &x,
18           &inc,&ret2, ...);
20      if(ret1 == 1) goto label;
21   }
22   label: ... ;
23   return ... ;
24 }
```

(b) example.c

```
1 #include "..."
2 typedef ... T;   (1)
3 int func_lp_1(int *i, int *x, int *inc,
4                   bool *ret2, ... );
5 int func_lp_2(int *i, int *x, int *inc,
6                   bool *ret2, ... );
```

note:

(a) is the original file,(b)(c)(d) are the new files created in CodeExtractor.

(1): user-defined type.

(2): global variable.

(3): jump statement.

(a) example_lp.h

```
1 #include "example_loop.h"
2 extern T *g;   (2)
3 int func_lp_1(int *i, int *x,
4          int *inc, bool *ret2, ...){
5    for((*i) = (*x), ; ... ; (*i) += (*inc)) {
6       ... = g;
7       if( condition1 )
8         {*ret2 = false; return 0;}   (3)
9       if( condition2 ){
10          ...; break; }
11   }
12   return -1; }
13 int func_lp_2(int *i, int *x, int *inc,
14                  bool *ret2, ... )
15 {
16   ...;//return  value  is  1
17 }
```

(a) example_lp.c

图 2　一个代码提取的例子

首先，由于代码片段被包装成新函数，并且放到新文件中，若要保证程序变换前后程序的一致性，需要考察程序片段中各种对象作用域的变化情况。对于待提取代码片段中的一个局部对象（在函数内且在该片段之前声明）来说，它在提取后的代码中会脱离其作用域的范围，从而导致程序编译错误以及语义的变化。因此，CCodeExtractor 在定位到待分割的代码片段之后，遍历代码片段中的各种对象，并将它们分成 5 类：变量（variable）、用户自定义类型（user-def）、跳转语句（jumps）、宏（Macro）、条件编译（cond-compile），其中变量又分为局部变量（var_l）、静态全局变量（var_{sg}）和普通全局变量（var_g），表 1 列出了它们的作用域（函数内、文件内或跨文件），详细处理细节将在下一节介绍。

表1 不同对象的作用域

作用域	变量			user-def	jumps	macros	cond-compile
	var_l	var_{sg}	var_g				
函数内	√	√	√		√	√	
文件内		√	√	√		√	√
跨文件			√	√			

其次,如果程序片段中出现了跳转语句(return,goto 等跳转语句的作用范围都在函数内部),其与待提取的代码片段放到新函数定义中时,将会脱离其作用域的范围,导致程序执行路径的错误。如何构造调用原型和新函数,并添加什么样的信息才能保证变换后程序执行路径的一致性,是 CCodeExtractor 代码提取中的一个关键问题。

最后,在函数调用原型和新函数建立完成之后,新函数被放到一个单独的文件中,怎样组织两个文件的共享信息(如头文件、共用的结构体等信息),关系到整个程序语义和编译的正确性。结合上面的分析,将 CCodeExtractor 代码提取的过程分为3个阶段:信息收集和处理、构造调用原型和新函数以及代码重组。

3 设计与实现

图3给出了 CCodeExtractor 在 LLVM[8] 上的处理框架,主要包含3个阶段:1)信息收集阶段(Information Collection)。遍历 Clang AST 结构,定位到符合指定条件的代码片段,然后将从 Clang[9] 编译器的预处理阶段和 AST 结构中收集与代码片段有关的信息并保存起来,包括变量、用户自定义类型(struct 或 union 等)、跳转语句(jump,goto 等)以及宏定义等信息。2)构造调用原型和新函数阶段(Construction of Calls and Functions)。根据第一阶段收集的信息,为不同形式的代码片段构造不同形式的函数调用原型和新函数等。3)代码重组阶段(Restruction of Source Code)。将构造完成后的代码重新组织并写入到对应的文件中。

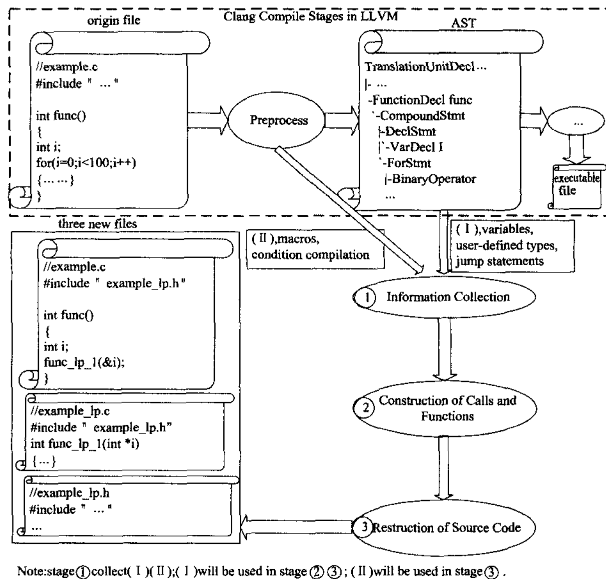

图3 CodeExtracor 的处理框架

3.1 信息的收集和处理

(1)变量

第2节从作用域的角度考虑了 CCodeExtractor 代码提取对不同对象作用域的影响,并将变量分为3类:普通全局变量 var_g、静态全局变量 var_{sg} 和局部变量 var_l。如表1所列,普通全局变量可跨文件使用,只须在使用的文件中用 extern 声明即可;静态全局变量只对本文件可见,不能跨文件使用;局部变量只对其声明的局部作用域可见。如果将待分割的代码片段包装成新函数放到其他文件中,某些类型变量也会随之脱离其作用域的范围。为了保证代码变换前后程序的一致性,CCodeExtractor 遍历代码片段中所有用到的变量并识别它的类型,将不同类型的变量保存起来,然后在构造参数时做相应的处理:静态全局变量和局部变量通过函数的参数传进新函数中;而普通全局变量不作为参数,在新生成的源代码文件中会使用 extern 重新声明。

(2)用户自定义结构

在一些大规模的程序中,程序员常常根据应用的特点在头文件或源文件中定义一些数据结构(struct,union 等类型),称为用户自定义类型。由于在 CCodeExtractor 函数提取中,新函数放到了单独的新文件中(图2 example_lp.c),如果使用到的用户自定义类型在头文件中定义,那么在新文件中直接包含这些头文件即可。若其定义在原文件,并且和待分割代码在同一个文件,则 CCodeExtractor 需要根据用户自定义类型找到其定义的内容(图2 example_old.c 中第2行的类型T),并在代码重组阶段将其定义移到新头文件中(图2 example_lp.h),否则,新函数中用户自定义类型的变量将找不到其类型的定义,导致编译出错。因此在遍历代码片段的同时,检查其类型是否是用户自定义的类型,并保存,以做后续处理。目前考虑的类型有 struct,union,enum 3种以及 typedef。

(3)跳转语句

跳转语句(goto,return,continue 等)作为C语言中的一类基本语句,在实际的编程开发中会经常用到,它们只能在函数内部实现跳转。当其随代码片段被提取到新函数中时,跳转语句的语义会发生相应的变化。

对于 goto 语句,因在不同条件下可能存在不同的跳转目标(label 语句),所以在变换时需要考虑如何保证原程序在新函数返回后跳转到正确的位置。在 CCodeExtractor 代码提取过程中,goto 及 label 的位置可能有3种情况:1)goto 和 label 都在待提取的代码片段内,这种情况不需要处理;2)待提取的代码片段中出现了 goto,但对应的 label 在该代码片段外定义,这种情况下先为 label 指定一个编号,然后将 goto 语句替换成 return 语句,返回值为对应 label 的编号,而在原程序文件中,会根据返回值(label 的编号)跳转到对应的 label;3)label在待分割的代码片段内,而对应的 goto 语句在该代码片段之外,目前 CCodeExtractor 不能处理这种情况。

对于 return 语句,跳转的目标是原函数的返回地址,当被提取到新函数中时,其跳转目标发生改变,会跳转到新函数的返回地址。在处理 return 时需要考虑两个问题:返回值如何传递和执行路径如何保证。CCodeExtractor 对 return 的处理如下:在原函数一侧,增加一个变量,将其地址作为参数,以便将真正的返回值传递回来,之后添加 return 将该新增的变量的值返回。在新函数一侧,将 return 的返回值赋给传进来的特殊参数,并将 return expr;语句替换成 return 0;,如图2的

example_old. c 的第 14 行的第一个 for 循环中 return false;语句，在 example. c 中的原函数 func 中添加了一个额外的变量 ret2 传递返回值，并在调用新函数后使用跳转语句 return ret2;返回真正的返回值；而在 example_lp. c 的新函数func_lp_1中，返回语句被替换成了 return 0;，并将返回值的表达式 false 赋给了一个参数 ret2 指向的单元(图 2 exampl_lp. c 的第 8 行)。

对于 break 和 continue 语句，它们经常和循环一块使用。CCodeextractor 在处理之前需要判断待分割的代码片段是否是原程序循环中的一部分，如果是，将其替换成 return －1;语句，并在原程序中添加一条 break 或 continue，否则语句不需要处理。

(4)条件编译与宏

在实际的应用程序开发中，为了方便程序的开发和修改，往往会增加一些宏和条件编译选项来控制程序中不同的处理策略。编译器在预处理阶段根据这些信息进行一些预处理操作(宏替换)，或者根据条件编译选项选择不同部分的代码模块来编译；在函数提取过程中，由于源代码片段在变换后的位置发生了改变，其所用到的宏信息等会由于相对位置的变动导致程序语义出现一些改变。如何在变换后的程序中修正相对位置变化带来的影响，是程序变换过程中必须要考虑的问题。

```
//original code
#include"head1. h"
#define Macro 50
#ifdef COND1

#include"header2. h"
  #define N 50
  int f1(){…}
#else
  #define N 100
  int a;
  int f1(){…}
  #endif
  int f2()
  {
  #undef Marco
  #define Marco 100
  for(…){
    //use Macro
     …}
  …
}
```

```
//file. c
#include"file_lp. h"

//—first part—
 #ifdef COND1
  int f1(){…}
 #else
  int a;
  int f1(){…}
 #endif


int f2()
{
#undef Marco
#define Marco 100
  f2_loop(…);
  …
}
```

```
//file_lp. h
#include"header1. h"
#define Macro 50
//—second part—
#ifdef COND1

#include"header2. h"
  #define N 50
#else
  #define N 100
#endif
```

```
//file_lp. c
#include"file_lp. h"
#undef Marco
#define Marco 100
void f1_loop(…)
{
  for(…){
    //use Macro
    …}
}
```

图 4　一个条件编译与宏的例子

图 4 给出了一个条件编译的例子，函数 f1 和 N 根据宏 COND1 的状态有不同的定义，函数 f2 先取消宏 Macro 的定义，然后又重新定义了 Macro。对于这一类代码，若要保证独立文件中的新函数使用到宏定义和头文件中的信息，须将它们放到公共的头文件 file_lp. h 中，但代码部分 first part 不应放到头文件中，因为当被多个文件包含时，则会出现重定义的错误。对于条件编译部分，CCodeExtractor 在原文件中找到其源代码部分，并将其分成两个部分：头文件和宏定义，以及代码部分(图 4 中 first part 和 second part)。而对于函数 f2 中宏的操作，为了保证循环中使用到的 Macro 的定义是 100，CCodeExtractor 从 Clang 预处理阶段收集宏的定义及宏取消的信息，然后在新文件(图 4 的 file_lp. c)中按照位置的先后顺序添加宏的操作(图 4 中的 file_lp. c 中增加了 Macro 取消和重新定义)。

3.2　构造调用原型与新函数

(1)构造调用原型

根据 2.1 节的分析，如果待分割的代码片段出现了跳转语句(goto，return 等)，为了保证执行路径不变，须对这些代码片段做一些处理，并在原程序中增加一些跳转语句。CCodeExtractor 根据跳转语句的类型，将待分割代码片段分为 5 类：Extract-normal，Extract-break，Extract-continue，Extract-return，Extract-goto。对所有类型的代码片段都会增加一个变量，用于接收新函数的返回值；新函数体内，从不同跳转语句返回，就会返回不同的值。表 2 列出了新函数代码片段的类别和新函数的返回值。

表 2　代码片段的分类

类型	Extract-normal	Extract-break	Extract-continue	Extract-return	Extract-goto
包含的跳转语句	无	break	continue	return	goto
新函数返回值	－1	－1	－1	0	＞0

Extract-return:

```
1. int ret_fn_original()
2. { …;
3.   for(i=1; i<100; i++){
4.        …;
5.        if(cond1)
6.             return exp1;
7.        if(cond2)
8.             return exp2;
9.   }
10.   …; }
```

```
1. int ret_fn()
2. {
3.   …;
4.   int ret1,ret2;
5.   ret1=loop_fun(&i,
6.              & ret2;…);
7.   if(ret1==0) return ret2;
8.   …;
9. }
```

```
1. int ret_fn_lp(int *i,int *ret2;
  …)
2. {
3.   for(*i=1; *i<100;
     *i++){
4.        …;
5.        if(cond1)
6.      { *ret2 = exp1; return
0;}
7.        if(cond2)
8.     { *ret2=exp2; return 0;
}
9.   }
10. return -1;
11. }
```

Extract-goto:

```
1. int goto_fn_original(){
2.   …;
3.   for(i=1; i<100;
     i++){
4.     …;
5.     if(cond1) goto label1;
6.     if(cond2) goto label2;
7.       …; }
8.   label1:…;
9.   label2:…;
10. }
```

```
1. int goto_fn(){
2.   …;
3.   int ret1;
4.   ret1=loop_fun(&i,…);
5.     if ( ret1  = = 1 ) goto
       label1;
6.     if ( ret1  = = 2 ) goto
       label2;
7.   label1:…;
8.   label2:…;
9. }
```

```
1. int goto_fn_lp(int *i,…)
2. {
3.   for( *i=1; *i<100;
     *i++){
4.     …;
5.     if(cond1) return 1;
6.     if(cond2) return 2;
7.     …;
8.   }
9.   return -1;
10. }
```

图 5　代码片段的处理策略

为了简化介绍，本节主要通过图 5 中的例子介绍后两种处理策略。处理 Extract-return 时，CCodeExtractor 为其增加了一个变量作为参数来传递真正的返回值(图 5 ret_fn 中的 ret2，在新函数中将所有原程序中 return 表达式的值赋给 ret2 指向的单元并作为参数传回，而 return 语句替换成 return 0;)，之后在调用语句后根据返回值增加一个 return 跳转语句；处理 Extract-goto 代码片段时，原函数在调用语句后根据返回值为每一个可能的 label 增加一条 goto 语句跳转到对应的 label。图 5 goto_fn 中为 label1 和 label2 各自增加了一条跳转语句。

最后，根据第一阶段收集到的信息构造实参列表，实参包含两类变量：信息收集阶段收集到的变量(局部变量、静态全局变量)和为传递返回值增加的变量(图 5ret_fn 中的 ret2)。CCodeExtractor 采用传递变量的地址值的方法，以保证原过程在调用新函数后能获取到变量修改后的值。

(2)新函数的构造

新函数的构造包含形参列表的构造和函数体的构造两部分。形参和实参列表包含的变量相同，但采取的形式不同，如 int 变量 a 在实参中为 &a，而在形参中为 int *a。待分割的代码片段是在信息收集阶段通过遍历 AST 树获取其开始和结束位置，然后从原文件中截取代码片段。函数体的构造主要是在待分割的代码片段上修改并增加一些信息：1)根据形参列表，将代码片段中对变量的使用改为对变量地址的间接使用，如对 a 的使用改为了(* a)；2)按照 2.1 节的规则处理跳转语句，这里只介绍 return 语句和 goto 语句的处理。对于 return 语句，将返回表达式赋值给一个传进来的参数，然后使用 return 0;从新函数返回，如图 5ret_fn_original 函数中 return expl；在新函数 ret_fn_lp 中被替换成了 * arg=expl; return 0;。而 goto 语句，在信息收集阶段，为每个 label 指定了一个编号，新函数中 goto 语句会被替换成 return 语句，返回值为 lebel 的编号，如图 5 函数 function2_orgingal 中 label1 和 label2 的编号分别为 1 和 2，新函数 goto_fn_lp 中 goto 语句替换成了 return 1;和 return 2;。

3.3 源代码的重组

在函数调用原型和新函数构造完成之后，需要重新组织源代码，以保证变换后的代码正确编译和变换前后程序语义的一致性。新函数定义放到一个单独的文件中，一方面便于程序员观察分割出来的程序片段的特点，另一方面对新函数的处理和执行更为方便(如单独编译或将新函数放到 GPU 等特殊计算部件上执行)。一个源代码文件经过 CCodeExtractor 处理后将会生成 3 个文件，例如文件 example_old.c 分割后生成 example.c，example_lp.c 和 example_lp.h。原文件中的内容经过处理后(头文件信息被提取，待分割的代码片段被替换成函数调用)写入到文件 example.c 中；example_lp.h 文件包含了头文件信息、结构体定义以及新函数的声明；使用 extern 声明的全局变量、新构造的函数以及相关的宏会按照位置的先后顺序写入到 example_lp.c。

4 实验评估

为了验证 CCodeExtractor 代码提取策略的有效性，基于 CCodeExtractor 代码提取策略在 LLVM3.3 上开发了一个提取程序中循环语句的工具，将符合分割条件(目前的分割条件是循环迭代的次数)的循环分离成单独的函数，再用这个工具进行测试。实验环境为：处理器 i7-3770 CPU@3.40GHz×4，内存 8GB，操作系统为 Ubuntu 12.04。

实验使用了 6 个不同规模的程序进行了测试，并通过指定条件分割迭代次数大于 10 次的循环(若迭代次数不能判断，默认分割)，表 3 给出了测试程序的名称以及程序中各种对象的信息：文件的个数 nfiles、代码行数 LOC、循环的总个数 nlps、提取的循环个数 ne-lps 以及提取的循环中各种对象出现的次数，others 包含了用户自定义类型和宏定义等信息。经过工具处理后，对比变换前后程序的运行结果，发现变换前后程序的运行结果是一致的，这也说明了 CCodeExtractor 适合于不同规模的程序。另外，在 Blas_d 测试程序上做了一个简单的实验，将迭代次数调整为 5 后，有 106 个循环被分割了。通过这个实验可以看出，CCodeExtractor 可以帮助我们在大规模程序中定位某些符合指定条件的代码片段(或不同类型程序片段，如 while 和 if 语句)，帮助理解和分析程序。

表 3 测试程序的特点

benchmark	Linpack_bench	Matmult	Blas_d	SIFT	tar-1.26	CLAPACK
nfiles	1	1	3	10	21	1583
LOC	1189	1974	5292	4456	19883	632289
nlps	25	40	107	197	138	4885
ne-lps	14	34	59	187	91	3373
ne-lps/nlps/%	56.00	85.00	55.14	94.92	54.17	96.05
locVar	75	151	431	988	346	13384
jmps	0	0	0	6	30	116
others	6	2	40	0	149	8

集成在 Eclipse 代码重构工具中的一个 C 源代码提取插件，通过人工选择一段代码并指定函数名，然后对代码片段不做任何处理直接由插件封装成新函数。通过对 Eclipse 的测试发现这种提取方法存在一些不足之处：1)变量地址虽然传到了新函数，但在新函数中的变量的使用形式没有变化，使原过程得不到变量修改后的值；2)不能处理包含跳转语句(如 return)的程序片段，因为跳转语句在被分割后会导致程序执行路径发生改变；3)没有处理预处理(宏、条件编译)信息导致变换后的程序不合法(图 1 变换后的程序 forfunc 中宏 N 未定义)；4)变换过程需要人工参与，不能自动化处理。但这些不能处理的代码片段会经常出现在程序中，如表 3 中最后 3

(下转第 29 页)

$\mu_2 \cap \{m_5\} = \emptyset \wedge (b_3' \times \{m_5\} \to b_4') \to Cal: \underline{\bot, b_4} \uparrow [\![CB_2 = b_4]\!] \to CB_2 = b_4 \wedge b_4' = F_2 \to \bot$。

结束语　本文从消息传递过程的角度研究了服务消息交互行为的元建模方法。首先基于工作流模型对服务进行建模；其次给出了接口相容性的检查方法；再结合推理规则和递归函数，描述与分析了服务交互行为的语义，并探讨了消息并行的情况；最后利用数据存储的实例对本文方法的有效性进行了说明。与此同时，我们在研究中还发现，结合规则和函数的形式化描述方法有助于刻画有监控需求的现实问题。下一步将会继续规范复杂服务交互过程，尝试把多消息接口检查问题转换到经典方法上对问题进行求解，并对多个服务间的交互验证做进一步的研究。

参 考 文 献

[1] TSAI W T, BAI X Y, HUANG Y. Software-as-a-service (SaaS): perspectives and challenges[J]. Science China Information Sciences, 2014, 57(5): 1-15.

[2] BROY M, KRÜGER I H, MEISINGER M. A formal model of services[J]. ACM Transactions on Software Engineering and Methodology, 2007, 16(1): 5.

[3] RICCOBENE E, SCANDURRA P. A formal framework for service modeling and prototyping[J]. Formal Aspects of Computing, 2014, 26(6): 1077-1113.

[4] BELKHIR W, CHEVALIER Y, RUSINOWITCH M. Parametrized automata simulation and application to service composition [J]. Journal of Symbolic Computation, 2015, 69: 40-60.

[5] CHEN S Z, FENG Z Y, WANG H. Service Relations and its Application in Services-oriented Computing[J]. Chinese Journal of Computers, 2010, 33(11): 2068-2083. (in Chinese)
陈世展，冯志勇，王辉. 服务关系及其在面向服务计算中的应用[J]. 计算机学报，2010，33(11)：2068-2083.

[6] WANG Z J, XU F, XU X F. Service network planning method for mass personalized functional requirements[J]. Journal of Software, 2014, 25(6): 1180-1195. (in Chinese)
王忠杰，徐飞，徐晓飞. 支持大规模个性化功能需求的服务网络构建[J]. 软件学报，2014，25(6)：1180-1195.

[7] PAPAZOGLOU M, GEORGAKOPOULOS D. Introduction to a special issue on service oriented computing[J]. Communication of ACM, 2003, 46(10): 25-28.

[8] SINGH M P, HUHNS M N. Service-oriented Computing: Semantics, Processes, Agents[M]. John Wiley & Sons, 2006.

[9] ORLOWSKA M E, WEERAWARNA S, PAPAZOGLOU M P, et al. Service-oriented Computing[C]// First International Conference Service-oriented Computing, Trento, Italy (ICSOC 2003). Preface, LNCS 2910, 2003.

[10] ZHANG C, DUAN Z H, TIAN C, et al. Modeling Verification and Test of Interactive Behaviors in Distributed Software Systems[J]. Journal of Computer Research and Development, 2015, 52(7): 1604-1619. (in Chinese)
张琛，段振华，田聪，等. 分布式软件系统交互行为建模，验证与测试[J]. 计算机研究与发展，2015，52(7)：1604-1619.

[11] SROKA J, HODDERS J, MISSIER P, et al. A formal semantics for the Taverna 2 workflow model[J]. Journal of Computer and System Sciences, 2010, 76(6): 490-508.

[12] ZHANG L J. EIC editorial: Introduction to the body of knowledge areas of services computing[J]. IEEE Transactions on Services Computing, 2008, 1(2): 62-74.

（上接第 20 页）

列给出了程序片段中这些程序在实际程序中出现的频率。CCodeExtractor 函数提取方法是通过指定的条件，在 AST 树上收集满足条件的代码片段，能够处理各种类别的代码。除了条件的指定外，变换过程不需要程序员的参与，实现了函数的自动化提取。

结束语　本文提出了一种自动化的代码提取的方法 CCodeExtractor，其将 C 程序中代码片段提取成单独的函数调用，一方面便于观察大规模程序局部片段的特征，另一方面能够方便程序的分析优化工作。另外，我们在 LLVM 上实现了一个循环提取的工具，实验表明该工具适用于不同规模的程序，并且能够进行正确的程序变换。

参 考 文 献

[1] FOWLER M. Refactoring: improving the design of existing code [M]. Pearson Education India, 1999.

[2] Coderefactoring[OL]. http://en.wikipedia.org/wiki/Code_refactoring.

[3] HAFIZ M, OVERBEY J. OpenRefactory/C: An infrastructure for developing program transformations for C programs[C]// Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity. ACM, 2012: 27-28.

[4] HAFIZ M, OVERBEY J, BEHRANG F, et al. OpenRefactory/C: An infrastructure for building correct and complex C transformations[C]// Proceedings of the 2013 ACM Workshop on Refactoring Tools. ACM, 2013: 1-4.

[5] Microsoft Visual Studio[OL]. http://www.wholetomato.com/features/feature-refactoring.asp#extract.

[6] Extract method In Eclipse[OL]. https://sourcemaking.com/refactoring/extract-method.

[7] JEFFREY L. Overbey, Farnaz Behrang, and Munawar Hafiz. A foundation for refactoring C with macros[C]// 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE). New York, NY, USA, 2014: 75-85.

[8] LATTNER C, ADVE V. LLVM: A compilation framework for lifelong program analysis & transformation[C]// International Symposium on Code Generation and Optimization, 2004 (CGO 2004). IEEE, 2004: 75-86.

[9] LATTNER C. LLVM and Clang: Next generation compiler technology[C]// The BSD Conference. 2008: 1-2.

[10] LLVM extractCode[OL]. http://llvm.org/docs/doxygen/html/CodeExtractor 8cpp _source.html.

[11] PROKAJ J. C code for SIFT feature point extraction[OL]. http://www.pudn.com/downloads93/sourcecode/graph/texture_mapping/detail365370.html.