Sistema de instrumentación estática de código fuente para mocking de funciones y objetos

Trabajo Final de Grado
Licenciatura en Ciencias de Computación

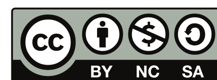
FAMAF - UNC

Autor: Andrés Tiraboschi

Director: Daniel Fridlender

Licencia

Esta obra está bajo una licencia Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional.



Resumen. Este trabajo consiste en el desarrollo de una herramienta de unit testing para el lenguaje de programación C.

Los principales problemas que poseen este tipo de herramientas es la intrusión en el código de producción y el impacto en el rendimiento del programa a la hora de testear. Este último punto hace que, por ejemplo en sistemas embebidos los test no sean lo suficientemente confiables ya que una pequeña diferencia en el rendimiento puede tener un gran impacto en el programa. Por ello, se busca desarrollar un sistema de test que utilice mocking, es decir que cree módulos que simulan el comportamiento de módulos complejos, y además:

- Sea plataforma independiente tanto de la arquitectura como del sistema operativo.
- Requiera cero intrusión del código fuente por parte de los usuarios.
- Mínimo impacto en desempeño de la aplicación (menor o igual al de otros sistemas de mocking como google mock, gmock).
- Provea absoluto control del comportamiento a mockear (en contraposición al conjunto de funcionalidades provisto por gmock).
- Que permita mockear tanto funciones definidas por el usuario como funciones de librerías externas (por ejemplo, las del sistema).
- Que su uso no implique grandes modificaciones en el build system del proyecto.

La herramienta desarrollada en este trabajo fue creada como un plugin de GCC y requirió introducir modificaciones en el compilador. Estás modificaciones fueron verificadas y aceptadas quienes están a cargo del desarrollo del compilador y ya vienen integradas desde la versión 6.0.0 de GCC. El parche "Plugin event for C/C++ function definitions" tiene como autor a Andrés Tiraboschi, fue aceptado y puede encontrarse en el siguiente link: https://patchwork.ozlabs.org/patch/473592/.

Índice

Capítulo 1. Introducción 1
1. Funciones mock en C y C++ 1
2. Mocking en sistemas embebidos 8

Capítulo 2. Sistema de mocking de funciones y objetos 11
1. Nociones preliminares 11
2. Diseño 20
3. Ejemplo 28

Capítulo 3. Conclusión 35
1. Trabajo futuro 35

Bibliografía 37

CAPíTULO 1

Introducción

1. Funciones mock en C y C++

El test unitario o prueba unitaria (unit-testing) es importante para el desarrollo de software: si se desea saber si el código funciona de la forma en que está diseñado, la mejor manera para obtener cierta certeza es probarlo, hacer testing. Sin embargo, el código puede ser bastante complejo. Podría llamar a muchas otras rutinas, y tratar con otras interfaces. Si es así, puede ser realmente difícil testear el código.

En el caso de que el código a ser testeado llame a funciones complejas que no queremos testear, podemos utilizar funciones stub para simplificar el testing. Es decir reemplazamos funciones complejas por funciones con la misma signatura, pero mucho más simples. Esta aproximación al testing sólo es factible de ser implementada en proyectos pequeños y no es conveniente por varias razones, la principal es que se contamina el código de manera poco manejable.

El mocking es algo diferente: cuando se está utilizando el código en producción no hay ninguna alteración de las funciones, pero cuando se hace la prueba unitaria en lugar de que cierto módulo llame a otros prediseñados, puede llamar a una interfaz "mock" que es mucho más simple. Luego, en el código que está siendo testeado, el módulo que simula (el módulo mock) puede interponerse cada vez que es llamado el módulo original. El problema es encontrar la mejor manera de hacer esto.

Para ser más concreto, lo que queremos hacer es convertir nuestro conjunto de funciones en un módulo para llamar a funciones mock en lugar de las verdaderas funciones.

Por ejemplo, si tenemos el siguiente código

```
int bar(int x)
{
  int result = 0;
  for (int i = 0; i < 10; i++)
  {
   result += foo(i);
  }
  return result;
}
```

Digamos que queremos testear la función `bar()` del código anterior. La función `foo()` que se utiliza en el código puede ser muy compleja, por ejemplo puede inspeccionar una base de datos, hacer cálculos intensivos, o algo que no queremos o no podemos testear. Entonces, para testear la función `bar()` es conveniente hacer una versión "mock" de `foo()` que sea mucho más simple que la original. Luego lo que necesitamos es alguna forma de reemplazar la llamada a `foo()`.

Una manera de hacer esto es usando un macro. Con un simple reemplazo de texto podemos hacer que `foo()` haga diferentes cosas, dependiendo si estamos o no haciendo el testing:

```
#ifdef TEST
#define FOO mock_foo
#else
#define FOO foo
#endif
int mock_foo(int x)
{
  return x;
}
int bar(int x)
{
  int result = 0;
  for (int i = 0; i < 10; i++)
  {
   result += FOO(i + x);
  }
  return result;
}
```

Ahora `bar()` llamará a `mock_foo` cuando `TEST` esté definido. Esto funciona pero no es del todo conveniente. El problema es que diferentes tests a la función `bar()` pueden necesitar diferentes formas de hacer el mocking de las funciones que son llamadas por `bar()`. La forma propuesta más arriba solo permite una forma de reemplazo, pero en general es deseable contar con una herramienta más flexible. Una manera de agregar flexibilidad es con el uso de un puntero a una función:

```
#ifdef TEST
#define FOO test_foo
#else
#define FOO foo
#endif
int (*test_foo)(int x) = foo;
int bar(int x)
{
  int result = 0;
  for (int i = 0; i < 10; i++) {
   result += FOO(i + x);
  }
  return result;
}
int mock_foo1(int x)
{
  return 0;
}
int mock_foo2(int x)
{
  return x;
}
int unittest(void)
{
  test_foo = mock_foo1;
  ASSERT(bar(3) == 0);
  test_foo = mock_foo2;
  ASSERT(bar(3) == 75);
  test_foo = foo;
}
```

Con este código podemos ahora hacer testing en forma más flexible. Sin embargo, esta técnica no carece de problemas. El primero es que la versión testeada del módulo no es la misma que la versión no testeada: la llamada se hace por un puntero a una función en vez de usar una función. La diferencia es pequeña pero podría causar que el compilador use diferentes registros y alocaciones afectando por ejemplo la performance. Existe la posibilidad que algunos bugs puedan manifestarse en el código “normal”, pero, debido a estas diferencias, no se disparen en el testing.

Otro problema es que, en general, al menos en el ámbito corporativo, el uso de macros no es recomendable. Observar que el código del testing difiere en varias líneas del código que sería utilizado en forma definitiva. Si sólo se hiciera mocking de unas pocas funciones esto no sería un problema, pero si es necesario hacer mocking de una cantidad considerable de funciones, entonces el código necesario para introducir esta herramienta de mocking será bastante diferente al código final. Lo óptimo sería que el código escrito por los programadores del proyecto no fuera alterado por los requerimientos de la prueba unitaria.

Existe otra forma para hacer mocking: podemos agregar una función mock que envuelve (“wrap”) las llamadas a `foo()`. La función wrapper puede testear un puntero de función para ver si debería llamar a la función original o a la función mock. Puesto que es una función y no un macro, los problemas de codificación desaparecen:

```
int bar(int x)
{
  int result = 0;
  for (int i = 0; i < 10; i++)
  {
   result += foo(i + x);
  }
  return result;
}
```

```
#ifdef TEST
int (*test_foo)(int x) = NULL;
int foo__(int x)
#else
int foo(int x)
#endif
{
  int result;
  /* algo complejo para calcular el resultado */
  return result;
}

#ifdef TEST
int foo(int x)
{
  if (test_foo) return test_foo(x);
  return foo__(x);
}
#endif
```

Luego se podría crear un archivo de test de la siguiente manera:

```
int mock_foo1(int x)
{
return 0;
}

int mock_foo2(int x)
{
return x;
}

int unittest(void)
{
  test_foo = mock_foo1;
  ASSERT(bar(3) == 0);
  test_foo = mock_foo2;
  ASSERT(bar(3) == 75);
  test_foo = foo;
}
```

Esto tiene diferentes ventajas y desventajas comparado con el método anterior. Una ventaja evidente es que `bar()` no se ve afectada por el testing. El problema ahora es que el código de `foo()` está "contaminado" por la necesidad de hacer mock. Usamos el wrapper más simple que se nos ocurre, sin embargo el header de la función está partido de una manera confusa. Existen otras maneras de hacer esto usando el pre-procesador de C, pero son complicadas y no recomendadas.

Ahora que hemos introducido mocking, vamos a revisar un framework concreto.

1.1. Google C++ Mocking Framework. Google ha desarrollado un framework de mock para el lenguaje de programación C++. El Google C++ Mocking Framework (o Google Mock, brevemente) es una librería para crear clases mock y utilizarlas. Se basa en el uso de macros y le dedicamos una subsección debido a su uso extensivo por los programadores de C++.

El uso de Google Mock requiere tres pasos básicos:

(1) usar algunos macros para describir la interfaz que se desea simular;
(2) crear algunos objetos mock y especificar sus espectativas y comportamiento con una sintáxis intuitiva.;

(3) ejercitar el código que utiliza los objetos mock. Google Mock detectará cualquier violación de las espectativas apenas se produzcan.

Google Mock permite evitar el uso de mocks manuales en C++, ayuda a eliminar dependencias innecesarias en los tests y hace que sean rápidos y confiables.

Los mocks hechos manualmente son propensos a tener errores, es por ello que en Java y Python los programadores cuentan con herramientas poderosas para realizar mocks. En el caso de Java, jMock y EasyMock son las herramientas más utilizadas. Por otro lado, Python incluye en su biblioteca estándar una librería para realizar mocking.

Google Mock fue diseñado para ayudar a los programadores de C++. Está inspirado en jMock y EasyMock, pero diseñado con el lenguaje C++ en mente. Su objetivo es lidiar con los siguiente problemas:

- pruebas lentas que dependen de muchas bibliotecas o usan recursos costosos (por ejemplo, una base de datos),
- pruebas frágiles que utilizan recursos no son confiables (por ejemplo, la red),
- probar cómo el código maneja un error (por ejemplo, un error de checksum del archivo), pero no es fácil causar uno (testear escenarios difíciles de alcanzar),
- interactuar con otros módulos de la manera correcta, cuando es difícil observar la interacción.

Google Mock está pensado como herramienta de diseño, pues permite experimentar con el diseño de la interfaz en etapas tempranas. También, obviamente, es una herramienta para el testing que permite interactuar con diferentes módulos en forma sencilla.

Google0 Mock propone dos escenarios posibles para mocking:

- **La clase real y la clase mock comparten una clase base.** Esto significa que si queremos simular el comportamiento de una clase existente en el código de producción y la misma no cuenta con una interfaz, requerirá de su declaración con sus funciones virtuales. Esto agrega una tabla virtual a nuestro diseño original.
- **La clase real y la clase mock no están relacionadas por una clase base.** En este caso que ambas contienen métodos con la misma signatura. Luego, habrá que decidir, en tiempo de compilación (y no en tiempo de ejecución como en el caso anterior), que versión se desea usar. Esto se da generalmente cuando una clase

que llamaremos 'Caller' utiliza a la clase real como parámetro de template. En este ejemplo 'Caller' en sus métdos invoca diferentes funciones de la clase real. En este caso tan solo basta con crear una clase mock con la misma signatura que la clase real las cuales no tienen por qué estar relacionadas entre si a través de una clase base. El problema es que para realizar la prueba unitaria la clase mock debe ser pasada como argumento de template y la resolución de templates se lleva a cabo en tiempo de compilación, por lo que se deberá decidir en tiempo de compilación que versión se desea utilizar.

2. Mocking en sistemas embebidos

El test unitario prueba un módulo de código aisladamente del resto del sistema. El módulo es típicamente una clase, una simple función o, posiblemente, un grupo de funciones relacionadas.

Los tests prueban a este módulo de acuerdo a su especificación y, para poder realizarlos, otras partes del sistema deben ser simuladas. Esto se hace, normalmente, como hemos visto en la sección anterior, mediante la implementación de funciones stub o por uso de mocking.

Una vez garantizado el éxito en las distintas pruebas unitarias, tendrán lugar las pruebas de integración para asegurar el correcto funcionamiento del sistema.

Una de las razones principales del presente trabajo es la necesidad de implementar casos de prueba unitarios eficientes en sistemas embebidos. Una de las dificultades que se afrontan al momento de trabajar en este tipo de entornos son las limitaciones en recursos. Las pruebas unitarias y los frameworks de mocks diseñados para el desarrollo de aplicaciones no embebidas cuentan con acceso a memoria virtual ilimitada. En contraste, un sistema embebido generalmente posee cantidades limitadas de memoria para minimizar su tamaño y el consumo de electricidad.

Por otra parte, para la utilización de mocking frameworks, en su mayoría, se requiere un alto grado de intrusión tanto en los módulos de testing como en el código de producción, como hemos explicado con el ejemplo de Google Mock.

El sistema a desarrollar tiene como finalidad mitigar los problemas mencionados previamente, por lo que deberá ser robusto, adaptable y eficiente.

El metodo propuesto en el presente trabajo satisface los siguientes requisitos:

1) Independiente de la plataforma tanto de la arquitectura como del sistema operativo.

Los sistemas embebidos están diseñados para cubrir un amplio rango de necesidades. En el mismo, la mayoría de los componentes se encuentran incluidos en la placa base (tarjeta de vídeo, audio, módem, etc.) Algunos ejemplos de sistemas embebidos podrían ser dispositivos como un taxímetro, un sistema de control de acceso, la electrónica que controla una máquina expendedora o el sistema de control de una fotocopiadora entre otras múltiples aplicaciones. Cada dispositivo utiliza su propia arquitectura, por lo que el sistema debe ser independiente de ella y del sistema operativo con el fin de garantizar el correcto funcionamiento. Permite ademas testear la aplicación en su entorno real, sin necesidad de simular su ambiente de operabilidad.

2) No requiere intrusión del código fuente por parte de los usuarios.

El sistema esta orientado a facilitar el uso y testing de los módulos de interés en un escenario mas cercano a su funcionalidad real. Es importante notar que la intrusión de código por parte del usuario para realizar alguna de las actividades mencionadas conlleva un alto costo de tiempo y a su vez puede introducir errores no deseados.

3) Mínimo impacto en desempeño de la aplicación (menor o igual al de otros sistemas de mocking como Google Mock).

En los sistemas de tiempo real, el testing debe ser capaz de representar casos de uso reales, en donde la performance y los tiempos de ejecución deben ser lo más cercano posible a la actividad final del modulo. Este hecho es de suma importancia para asegurar el correcto funcionamiento del modulo, por lo que la performance no debe ser alterada al momento de ejecutar los distintos test unitarios.

4) Utilizable para programas escritos en C.

El sistema final se puede utilizar para testear programas escritos en C, uno de los lenguajes habituales de los sistemas embebidos.

La herramienta desarrollada en este trabajo fue implementada como un plugin de GCC y escrita en C++. Su implementación requirió modificar GCC con el fin de introducir nuevos eventos de plugin. Un parche

cuyo autor es Andrés Tiraboschi fue enviado a los mantenedores de GCC para su verificación, fue aceptado y forma parte de GCC a partir del release 6.0.0. El parche puede ser encontrado en el siguiente link: http://permalink.gmane.org/gmane.comp.gcc.patches/342255.

CAPíTULO 2

Sistema de mocking de funciones y objetos

1. Nociones preliminares

En esta sección se explicarán los conceptos básicos tales como mocking, qué es GIMPLE, qué es un plugin de gcc, etc. Estas nociones fueron componentes fundamentales para poder realizar el trabajo.

1.1. GCC. Por completitud, explicaremos brevemente nociones básicas sobre el funcionamiento interno de este compilador, pues el trabajo se basa en gran parte en el uso de GCC.

Es bien conocido que un compilador es un programa que, dado un código en un lenguaje de programación, lo procesa y lo convierte en otro lenguaje (por lo general en assembler).

El GNU Compiler Collection (GCC) es un conjunto de compiladores producido por el proyecto GNU que soporta varios lenguajes de programación. GCC es un componente clave de la cadena de herramientas GNU y es el compilador estándar para la mayoría de los sistemas operativos tipo Unix. La Free Software Foundation (FSF) distribuye GCC bajo la Licencia Pública General GNU (GNU GPL). GCC ha jugado un papel importante en el crecimiento del software libre, como una herramienta y un ejemplo.

Originalmente llamado GNU C Compiler, cuando sólo manejaba el lenguaje de programación C, GCC 1.0 fue lanzado en 1987 y fue ampliado para compilar C++ en diciembre de ese año. Posteriormente, fueron desarrollados front ends para Objective-C, Objective-C++, Fortran, Java, Ada y Go, entre otros.

GCC ha sido portado a una amplia variedad de arquitecturas de procesador, y se usa como una herramienta en el desarrollo de software libre y propietario. GCC también está disponible para la mayoría de los sistemas embebidos, incluyendo ARM, AMCC y chips de arquitectura Freescale Power.

Además de ser el compilador oficial del sistema operativo GNU, GCC ha sido adoptado como compilador estándar por muchos otros sistemas operativos modernos como Unix, incluyendo Linux y la familia BSD, aunque FreeBSD y macOS se han trasladado al sistema LLVM. También hay versiones disponibles para Microsoft Windows y otros sistemas operativos.

A la hora de compilar GCC divide este proceso en diferentes etapas. Podemos resumir las etapas de la compilación en lo siguiente (ver Figura 1.1):

- **front-end:** es la etapa donde se llevan a cabo todas las tareas de parseo y de preprocesamiento. En este momento se generan todas las estructuras necesarias para el middle-end. Pese a que la mayor parte de las optimizaciones se realizan en el middle-end hay excepciones tales como 'constexpr' que se llevan a cabo durante el parseo.

  La etapa de parseo genera como salida GENERIC, que es una forma de representación común para todos los lenguajes soportados por GCC. En esta representación todas la funciones están descriptas en forma de árboles. El hecho de que esta representación sea independiente del lenguaje tiene el fin de que el resto de la compilación sean también independiente del lenguaje, con lo cual no es necesario desarrollar un middle-end y un back-end por cada lenguaje soportado. El front-end de GCC genera GENERIC, la cual es bajada luego a otra representación basada en árboles, llamada GIMPLE y luego a RTL.
- **middle-end:** al igual que GENERIC existen otros lenguajes de representación intermedia tales como GIMPLE y RTL. Estos dos últimos son estructuras propias del middle-end del compilador y tienen como fin realizar optimizaciones sobre el código y generar lo que finalmente será el input del back-end.

  GIMPLE permite de llevar a cabo las optimizaciones que son independientes del lenguaje tales como inlining, propagación de constantes, eliminación de redundancia, etc. Pese a que tanto GIMPLE como GENERIC son independientes del lenguaje, su principal diferencia entre ellos es que GIMPLE es más restrictivo ya que no permite que las expresiones posean más de 3 operandos, a excepción de las funciones que pueden tener una cantidad arbitraria de

argumentos, no consta con estructuras de flujo de control y las expresiones con efectos secundarios solo pueden estar del lado derecho de las asignaciones.

Luego que todas las pasadas de GIMPLE fueron llevadas a cabo se genera el RTL donde se realizarán más optimizaciones tales como optimizaciones de loops. RTL será el output del middle-end y el input del back-end del compilador donde finalmente se convertirá a assembler.

- **back-end:** en esta fase a partir del RTL proveniente del middle-end se genera el assembler que finalmente será ejecutado por la máquina.

1.2. Mocking. Mocking, como fue explicado en la introducción, es una técnica de testing unitario utilizada sobre todo en programación orientada a objetos. Esta técnica consiste en crear objetos que simulen determinados comportamientos especificados por el usuario con el fin de simular diferentes escenarios de falla.

Esta técnica es de utilidad cuando se desea simular el comportamiento de objetos complejos facilitando de gran manera el trabajo en comparación de que si se trabajara con los objetos reales.

Otra ventaja del mocking es el testing de objetos interdependientes, en donde por ejemplo se desea testear el funcionamiento de un objeto A que depende de un objeto B que aún no está implementado simplemente proveyendo una implementación básica de la interfaz de B.

El uso de esta técnica es de utilidad en las siguientes situaciones:

- El objecto a simular posee métodos que retornan resultados no determinísticos. En este caso, de usarse el objeto real, podría llegar a ser sumamente difícil reproducir casos en donde es importante que el resultado de retorno de un cierto método esté en un rango dado.
- El objeto a simular posee estados que son difíciles de alcanzar, como por ejemplo estados a cuya alcanzabilidad requiera de muchas variables de otros objetos con los que se esté interactuando o errores de conexión o de un motor de base de datos, etc.
- Casos donde la inicialización del objeto simulado tome demasiado tiempo como podría ser la inicialización de una base de datos. De no usar esta técnica, a la hora de correr los test unitarios se perdería mucho tiempo.

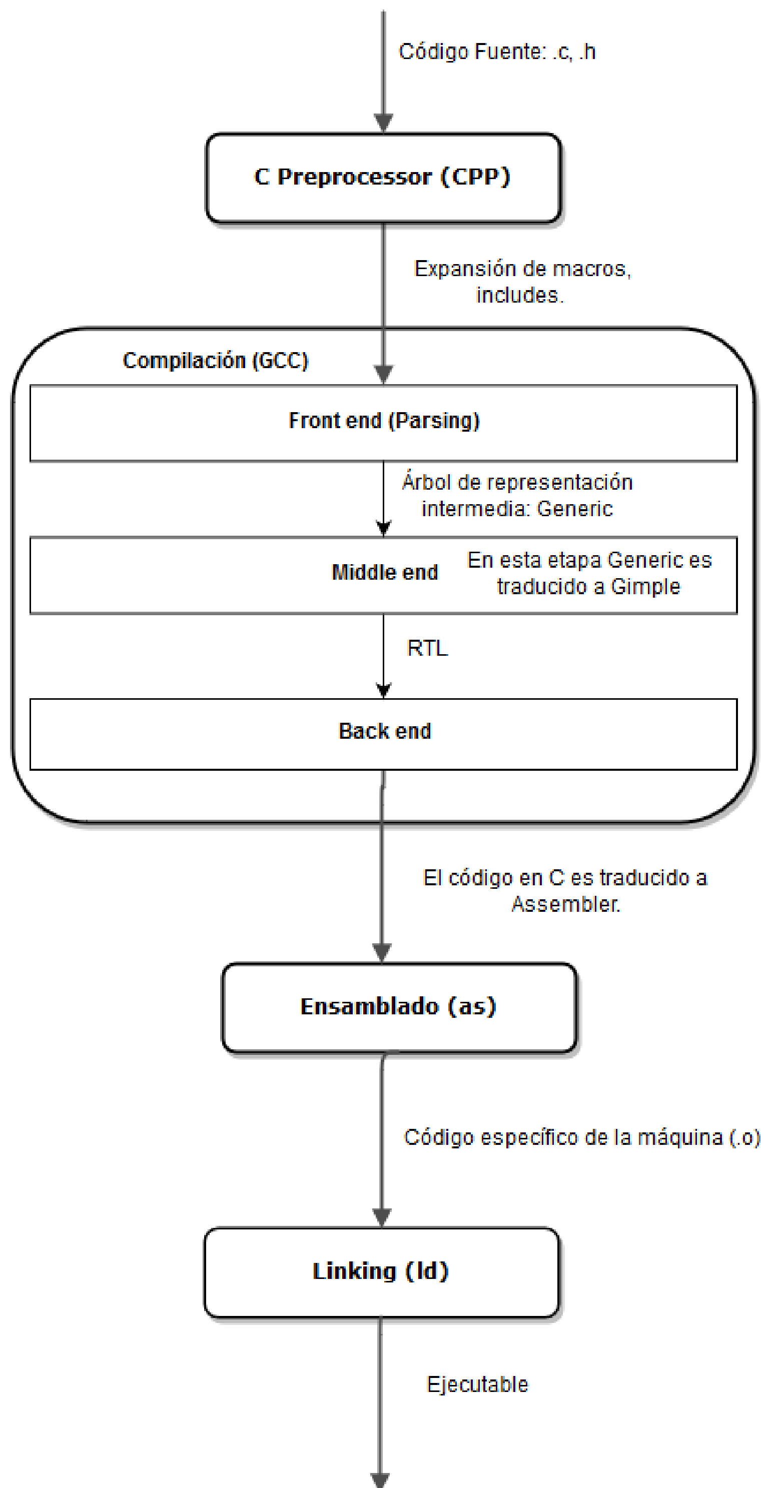


Figura 1. Gráfico de las etapas de compilación.

- Si en el objeto simulado se desean incluir métodos o atributos propios del testing y no de la tarea específica que debe desempeñar el objeto real.

Una desventajas de esta técnica es por ejemplo el hecho de que la interacción de objetos reales con objetos simulados requiere modificar el código original, que es costoso a nivel de tiempo y puede introducir nuevos bugs.

Otro problema es el hecho de que al realizar el testing de una aplicación de esta manera, la performance del programa se altera, por lo que el escenario de testing no sería el real. Esto es de vital importancia en entornos de desarrollo de sistemas embebidos donde el hardware puede llegar a ser limitado o en sistemas de tiempo real donde la performance también es de vital importancia.

Esta técnica también es útil en paradigmas de programación no orientados a objetos donde en vez de simular objetos se pueden simular funciones.

1.3. Prepocesador. En ciencias de la computación, un preprocesador es un programa que procesa datos de entrada para producir salida que se utiliza como entrada a otro programa. La salida se dice que es una forma preprocesada de los datos de entrada, que es a menudo utilizado por algunos programas posteriores como compiladores. La cantidad y tipo de procesamiento realizado depende de la naturaleza del preprocesador; algunos preprocesadores sólo son capaces de realizar sustituciones textuales relativamente simples y expansiones de macro, mientras que otros tienen el poder de lenguajes de programación completos.

Un ejemplo común de programación de computadoras es el procesamiento realizado en código fuente antes del siguiente paso de compilación. En algunos lenguajes informáticos (por ejemplo, C y PL/I) hay una fase de traducción conocida como preprocesamiento. También puede incluir procesamiento de macros, inclusión de archivos, extensiones de lenguaje y preprocesamiento racional.

Tanto en C como en C++, el preprocesador es el encargado de procesar directivas tales como macros, inclusión de archivos (`#include`, condicionales (`#if`, `#elif`, `#endif`, `#ifdef`, `#ifndef`), pragmas (`#pragma`), control de lineas (`#line`), etc.

En primer lugar el preprocesador se encarga de reemplazar los trigrafos por los caracteres que representan y se unen las lineas terminadas en backslash. Luego el resultado de lo anterior es descompuesto en tokens de preprocesador y los comentarios son eliminados.

Posteriormente se ejecutan las directivas del preprocesador como por ejemplo la invocación de macros y la inclusión de archivos (`#include`).

La directiva `#include` funciona indicándole al preprocesador que reciba el input desde el archivo especificado, antes de continuar con el actual. Como resultado del preprocesador se tendrá el output generado antes de `#include`, seguido por el output correspondiente al archivo especificado y por último el output de lo que está a continuación de `#include`.

Luego de esto el preprocesador debe encargarse de procesar las “escape sequences”, como por ejemplo `\n` que se traduce en fin de linea, en strings y caracteres constantes y se concatenan los string literals adyacentes.

1.4. Plugins de GCC. Los plugins del compilador permiten que un desarrollador añada nuevas características al compilador sin tener que modificar el compilador en sí mismo. Los plugins para GCC están disponibles desde la versión 4.5.0 y posterior. Las funcionalidades de los plugins ofrecen varias ventajas:

- Reducen el tiempo necesario para crear y probar nuevas características. Sólo es necesario compilar el código necesario para implementar la nueva funcionalidad.
- Permiten el desarrollo de características del compilador que por una razón u otra no son adecuadas para su inclusión en la distribución principal de GCC.
- Simplifica el trabajo de los desarrolladores que necesitan modificar GCC, pero no tienen el tiempo o la inclinación de profundizar demasiado en la parte interna del compilador.

Un plugin de GCC debe ser implementado en C++ y permite ejecutar código definido externamente al de GCC que puede ser útil como, por ejemplo, realizar análisis estático de código, y agregar nuevas funcionalidades a la compilación tales como optimizaciones u otras transformaciones de código.

GCC provee una API para plugins que consta de eventos, los cuales en el momento en que ocurren, llaman a una callback definida por el usuario. En esta función el usuario tiene la posibilidad de trabajar con los parámetros que se pasan con la posibilidad de realizar análisis, etc. Los plugins permiten realizar esto en etapas de la compilación tales como: front-end, middle-end y back-end.

Los Eventos de plugin que ofrece GCC son los siguentes:

```
enum plugin_event
```

```
{
PLUGIN_PASS_MANAGER_SETUP,    /* To hook into pass manager.  */
PLUGIN_FINISH_TYPE,           /* After finishing parsing a type.  */
PLUGIN_FINISH_DECL,           /* After finishing parsing a declaration. */
PLUGIN_FINISH_UNIT,           /* Useful for summary processing.  */
PLUGIN_PRE_GENERICIZE,        /* Allows to see low level AST in C and C++ frontends.
PLUGIN_FINISH,                /* Called before GCC exits.  */
PLUGIN_INFO,                  /* Information about the plugin. */
PLUGIN_GGC_START,             /* Called at start of GCC Garbage Collection. */
PLUGIN_GGC_MARKING,           /* Extend the GGC marking. */
PLUGIN_GGC_END,               /* Called at end of GGC. */
PLUGIN_REGISTER_GGC_ROOTS,    /* Register an extra GGC root table. */
PLUGIN_REGISTER_GGC_CACHES,   /* Register an extra GGC cache table. */
PLUGIN_ATTRIBUTES,            /* Called during attribute registration */
PLUGIN_START_UNIT,            /* Called before processing a translation unit.  */
PLUGIN_PRAGMAS,               /* Called during pragma registration. */
/* Called before first pass from all_passes.  */
PLUGIN_ALL_PASSES_START,
/* Called after last pass from all_passes.  */
PLUGIN_ALL_PASSES_END,
/* Called before first ipa pass.  */
PLUGIN_ALL_IPA_PASSES_START,
/* Called after last ipa pass.  */
PLUGIN_ALL_IPA_PASSES_END,
/* Allows to override pass gate decision for current_pass.  */
PLUGIN_OVERRIDE_GATE,
/* Called before executing a pass.  */
PLUGIN_PASS_EXECUTION,
/* Called before executing subpasses of a GIMPLE_PASS in
execute_ipa_pass_list.  */
PLUGIN_EARLY_GIMPLE_PASSES_START,
/* Called after executing subpasses of a GIMPLE_PASS in
execute_ipa_pass_list.  */
PLUGIN_EARLY_GIMPLE_PASSES_END,
/* Called when a pass is first instantiated.  */
PLUGIN_NEW_PASS,
/* Called when a file is #include-d or given via the #line directive.
```

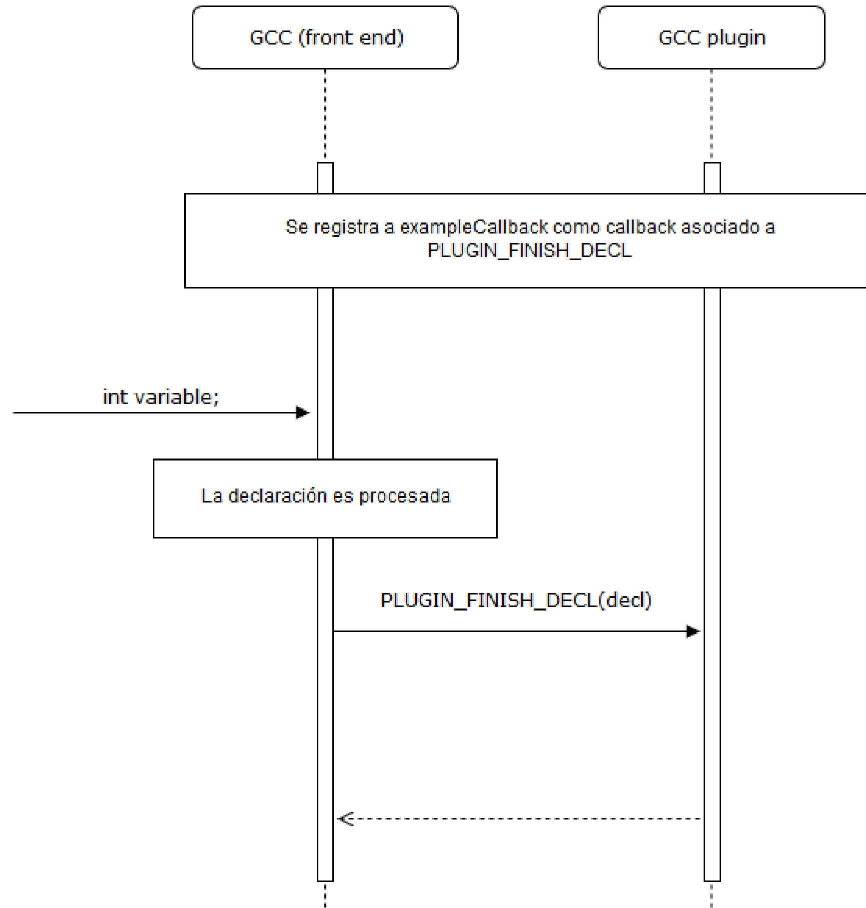


FIGURA 2. Ejemplo de diagrama de secuancia de un evento de plugin.

```
This could happen many times.  The event data is the included file path,
as a const char* pointer.  */
PLUGIN_INCLUDE_FILE,

PLUGIN_EVENT_FIRST_DYNAMIC    /* Dummy event used for indexing callback
array.  */
};
```

Por ejemplo el evento `PLUGIN_FINISH_DECL` será invocado cuando se haya terminado de parsear la declaración de una variable.

Algunos ejemplos de plugins de GCC son:

- ODB: ODB es un object-relational mapping (ORM) system para C++. Permite persistir objetos de C++ en una base de datos

relacional sin la necesidad de tener que lidiar con tablas, columnas o SQL.
- gcc-vcg-plugin: Este plugin permite imprimir estructuras internas de GCC de forma gráfica a la hora de debuggear el compilador.
- MELT: framework para crear plugins de middle-end para crear nuevas pasadas o para análisis a través de un lenguaje de alto nivel similar a Lisp.

1.6. GIMPLE. Los plugins desarrollados en este trabajo actuarán directamente en GIMPLE. Por lo tanto, pese a que fue mencionado anteriormente, nos extendermos sobre esta etapa de compilación.

GIMPLE es una forma de representación intermedia del código, propia del middle-end, la cual proviene de descomponer las estructuras de GENERIC a tuplas de no más de tres elementos a excepción de las tuplas que representan llamados a funciones.

GIMPLE se obtiene a partir de GENERIC traduciendo estructuras de control de datos complejas como loops, funciones anidadas y condicionales entre otros a jumps condicionales y finalmente traduciendo el conjunto de las distintas expresiones del programa a una forma SSA. La etapa por la cual GENERIC es traducido a GIMPLE es llamada 'gimplifier'.

En GIMPLE existen diferentes tipos de sentencias. En este trabajo le daremos importancia a uno: `GIMPLE_CALL`. Las sentencias de este tipo caracterizan llamados a funciones y vienen dadas en forma de tuplas de la siguiente manera:

```
[Fn, Lhs, a1, a2, ... , an]
```

Donde:

- `Fn` es la representación de la función que se está llamando.
- `Lhs` la representación de la variable a la cual se le está asignando el valor de retorno de la función. En caso de que la función no retorne nada o de que su valor de retorno no sea asignado a ninguna variable el valor de Lhs será NULL.
- `a's` son las representaciones de los argumentos de la función.

Por ejemplo la siguiente expresión:

```
var = function(arg1, arg2, arg2)
```

En GIMPLE será caracterizada de la siguiente manera:

```
[rep(function), rep(var), rep(arg1), rep(arg2), rep(arg3)]
```

donde `rep(x)` es la representación interna del compilador de la expresión `x`.

2. Diseño

2.1. Plugin front end. Este modulo es el encargado de inyectar código y realizar las modificaciones necesarias en la etapa de parseo del archivo a compilar. Por ejemplo cuando se encuentre una declaración o definición relevante con el fin de realizar las transformaciones necesarias.

2.1.1. *Reglas de transformación.* Dada la función `functionToMock` a la cual se desea mockear se pensó en la siguiente regla de transformación:
CASO 1: se encuentra la definición de `functionToMock`.

```
Type1
functionToMock(T1 t1, ... , Tn tn)
{
  A
}
```

Se transforma el código, en primer lugar, modificando el nombre de la función `functionToMock` a `functionToMock_ORIGINAL`. En segundo lugar se introduce la declaración de un puntero a función llamado `functionToMock_PTR` del mismo tipo que `functionToMock_ORIGINAL`. Por último se introduce una nueva definición de función llamada `functionToMock`, con el mismo nombre y prototipo que la función original, de la siguiente manera:

```
Type1 functionToMock_ORIGINAL(T1 t1, ... , Tn tn)
{
   A
}

(Type1 *)functionToMock_PTR(T1 t1, ... , Tn tn) = functionToMock_ORIGINAL;

Type1 functionToMock(T1 t1, ... , Tn tn)
{
  return functionToMock_PTR(t1, ... , tn);
}
```

de esta manera modificando `functionToMock_PTR`, es posible hacer que `functionToMock` se comporte de una manera arbitraria asignándole cualquier otra función con el mismo prototipo inicializada en `functionToMock_ORIGINAL`. Observar que si el valor del puntero a función `functionToMock_PTR` es `functionToMock_ORIGINAL` el programa se comportará de la misma manera que en su versión original.

CASO 2: se encuentra una declaración de 'functionToMock':

```
Type1 functionToMock(T1 t1, ... , Tn tn);
```

En este caso simplemente se introducirán las declaraciones del puntero a función `functionToMock_PTR` y de la función `functionToMock_ORIGINAL`. La semántica del programa resultante será la misma que la del siguiente código:

```
Type1 functionToMock(T1 t1, ... , Tn tn);
(Type1 *)functionToMock_PTR(T1 t1, ... , Tn tn);
Type1 functionToMock_ORIGINAL(T1 t1, ... , Tn tn);
```

Este mecanismo sin embargo posee una limitación, que es el que esto no es posible de realizar para funciones de librería, es decir funciones que son linkeadas externamente. Un ejemplo de esto sería en el caso de que se quiera probar el comportamiento de una función que un punto desee reservar memoria utilizando 'malloc' y que por algún motivo esto falle. No es posible realizar mocking sobre malloc ya que la definición de esta función no se encuentra en el código del programa. Dado esto se pensó en otra regla de transformación para estos casos.

Sea el siguiente código perteneciente a la declaración de una función de librería llamada `libraryFunction`:

```
Type1 libraryFunction(T1 t1, ... ,Tn tn);
```

Dada esta declaración, al igual que en la regla de transformación anterior se introducirá la declaración del puntero a función `libraryFunction_PTR` y en segundo lugar la definición de una nueva función llamada `libraryFunction_MOCK`.

La semántica de la compilación del código anterior será equivalente a la del siguiente código.

```
Type1 libraryFunction(T1 t1, ... ,Tn tn);


(Type1 *)libraryFunction_PTR(T1 t1, ... ,Tn tn);
```

```
Type1 libraryFunction_MOCK(T1 t1, ... ,Tn tn)
{
  return libraryFunction_PTR(t1, ... , tn);
}
```

Observar que si la declaración de `libraryFunction` ocurre más de una vez en la unidad de traducción se producirá un error de compilación ya que se tendrían definiciones múltiples de `libraryFunction_MOCK`. Por este motivo esta transformación solo se realizará en la primera ocurrencia de `Type1 libraryFunction(T1 t1, ... ,Tn tn);` y se evitará este problema.

Con esta transformación en principio si se modifica el valor de `libraryFunction_PTR` el comportamiento del programa original con el modificado debería ser el mismo ya que en principio `libraryFunction_MOCK` no es invocada en ningún lugar del programa. Para lograr el mismo resultado que en la primer regla descripta, fue necesario que en la etapa de representación intermedia de `gcc GIMPLE`, se reemplacen los llamados a `libraryFunction` por llamados a `libraryFunction_MOCK`. De esta forma modificando el valor de `libraryFunction_PTR` es posible realizar que `libraryFunction` se comporte de acuerdo a como lo especifique el usuario. Esta transformación en la etapa de middle end del compilador será detallada en la siguiente subsección.

En la primera regla de transformación fue necesario modificar el nombre de una función. Para lograr esto se requirió introducir un nuevo evento de plugin `START_PARSE_FUNCTION` en la función interna de gcc encargada de parsear las definiciones de funciones. Cuando ocurre este evento se invoca una callback que se encarga de modificar el nombre de la función en la representación interna de gcc. Para conseguir saber dónde introducir código fue necesario crear un nuevo evento de plugin llamado `END_PARSE_FUNCTION`. Este evento es invocado cuando el compilador finaliza con el parseo de una definición función. Cuando esto sucede, es llamado un callback del plugin que analiza las estructuras internas del compilador con el fin de determinar cuál función es la que se ha terminado de parsear.

Con el fin de conocer que declaraciones de funciones han sido parseadas se utilizó el evento de plugin ya existente `FINISH_DECL`.

Tanto los eventos `START_PARSE_FUNCTION` y `END_PARSE_FUNCTION` con sus respectivos callbacks fueron aceptados por la comunidad y ahora forman parte de gcc. En el siguiente link se puede encontrar el parche: http://permalink.gmane.org/gmane.comp.gcc.patches/342255.

Tanto en la primera como en la segunda regla fue necesario inyectar nuevo código al ya existente. En la representación interna de gcc el tipo de dato que provee la abstracción de un archivo es `cpp_reader`.

De forma preliminar se inserto código alterando el proceso de lectura del archivo, generando una `cpp_reader` modificado.

Esto, a fines prácticos, es inútil dado que para tener conocimiento donde introducir código se debería ser capaz de parsear el archivo para detectar declaraciones y definiciones, lo cual supondría una gran cantidad de trabajo con mucha probabilidad de introducir errores dada la complejidad del lenguaje. Esto sería básicamente realizar el mismo trabajo que realiza el parser del compilador. Por esta razón esta idea fue descartada.

A continuación lo que se intentó fue, a partir de un archivo generado por el plugin, crear un `cpp_reader` falso.

En caso de que en la etapa de parseo se encuentre una declaración o definición relevante, el estado actual del parser es guardado al igual que el `cpp_reader` que está procesando, el cual es reemplazado por el `cpp_reader` que fue generado artificialmente.

Luego cuando el parser finaliza con el `cpp_reader` provisto artificialmente, el estado y el `cpp_reader` originales son restaurados. Posteriormente de esto, el proceso de parseo continua normalmente.

Esta alternativa funcionó de manera satisfactoria pero requería de numerosos cambios al código de gcc por lo que se consideró que no era la mejor alternativa ya que era probable que estas modificaciones no fueran aceptadas por la comunidad.

Por último se terminaron utilizando primitivas pertenecientes al preprocesador de gcc, logrando emular la inclusión de un archivo.

Gracias a esto, al encontrar una declaración o definición relevante, se genera un archivo `tmock.inc` donde se escribe el código que se desea insertar y luego este archivo es incluido en el punto donde se desea introducir código. De esta forma se logra introducir las declaraciones de variables o funciones que sean necesarias para transformar el programa de la forma descripta anteriormente.

Al insertar código de esta manera el mismo es introducido justo en el punto del código donde se encuentra leyendo el parser.

Este método no necesitó de modificaciones importantes al código del compilador por lo que fue considerado como la mejor opción.

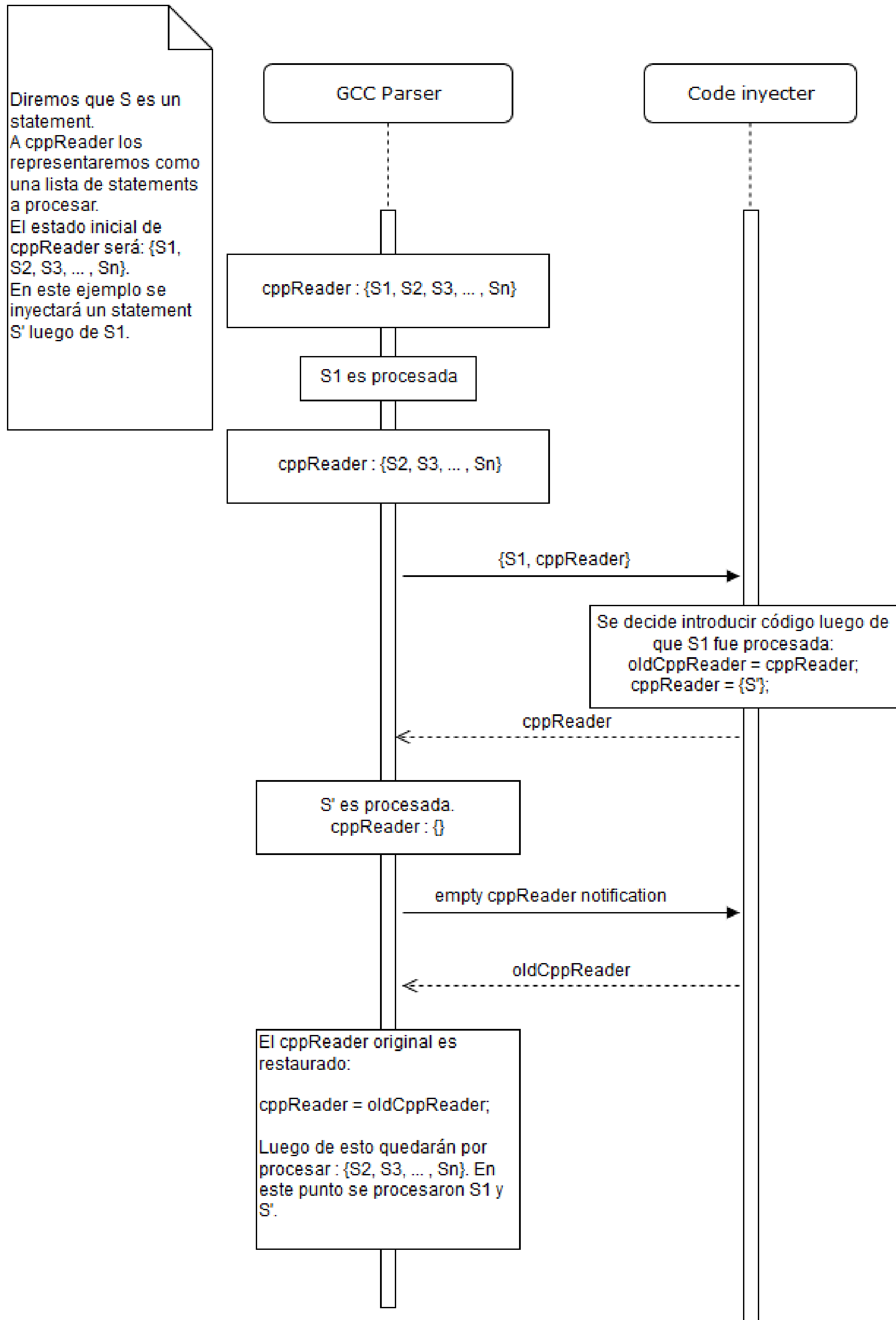


Figura 3. Diagrama de secuencia.

Por ejemplo en la segunda regla de transformación la introducción de código se realiza de la siguiente manera:

Dado el archivo `file.c`:

```
.
.
.

Type1 libraryFunction(T1 t1, ... ,Tn tn);

.
.
.
```

se creará un nuevo archivo `tmock.inc`:

```
(Type1 *)libraryFunction_PTR(T1 t1, ... ,Tn tn);

Type1 libraryFunction_MOCK(T1 t1, ... ,Tn tn)
{
  return libraryFunction_PTR(t1, ... , tn);
}
```

y utilizado primitivas del preprocesador el parser terminará procesando lo siguiente:

```
.
.
.

Type1 libraryFunction(T1 t1, ... ,Tn tn);

include "tmock.inc"

.
.
.
```

Se puede observar que todo lo que puede ser realizado con la primera regla también puede ser realizado utilizando la segunda. Por otra parte

se puede ver que tanto la primera como la segunda regla poseen la misma cantidad de indirecciones, una indirección, a la hora de realizar el llamado a la función, por lo que su impacto en el rendimiento es el mismo.

Además de que la segunda regla de transformación puede trabajar con todo tipo de funciones también existe la posibilidad no de reemplazar todos los llamados a la función original por la creada artificialmente en la etapa de middle end. Esto posibilita testear el funcionamiento de la función a la cual se desea reemplazar en puntos específicos de códigos, por ejemplo en ciertas funciones. Esto posibilita también realizar casos de test más específicos.

Por estas razones se decidió descartar la primera regla y quedarse solo con la segunda. Esto también es positivo ya que el usuario no debe preocuparse al momento de que regla de transformación utilizar.

2.2. Plugin middle end. Este es el módulo encargado de realizar los reemplazos de una función por otra en la etapa de representación intermedia de la compilación. Este módulo al encontrar un llamado a función, verifica si la función que se está llamando requiere ser reemplazada. En ese caso modificará ese llamado a función de forma tal que la función que se invoque sea la creada artificialmente por el plugin.

Para lograr esto fue necesario trabajar en la etapa de representación intermedia del compilador. Más precisamente en la etapa de GIMPLE, justo después de que el programa fue traducido a SSA. En esta etapa es posible, por cada definición de función, analizar y modificar el cuerpo de cada una. En nuestro caso solo son relevantes los llamados a funciones. Estos llamados están representados en forma de tuplas de la siguiente manera:

Sea `rep(X)` la representación interna de `X`. Por ejemplo si `fun1` es una función, `rep(fun1)` será la representación interna de esa función. Sea `fun1` una función en cuyo cuerpo se invoca a `fun2`:

```
int fun2(int a, int b)
{
  return a + b;
}

int fun1(void)
{
  int x;
  int y;
  int ret;
```

```
  x = 3;
  y = 4;
  ret = fun2(x, y); //1
  return ret;
}
```

En GIMPLE la linea 1 se representa de la siguiente manera:

```
(GIMPLE_CALL rep(fun2), rep(ret), rep(x), rep(y))
```

Si por ejemplo se desea reemplazar los llamados fun2, luego de que el módulo verifique que en el llamado a función se está invocando a fun2, se realizará el reemplazo de fun2 por fun2_MOCK. Posteriormente la linea 1 del ejemplo anterior quedaría representada de la siguiente manera:

```
(GIMPLE_CALL rep(fun2_MOCK), rep(ret), rep(x), rep(y))
```

De esta manera se logra, siempre que se desee, que sea llamada la función fun2_MOCK en lugar de fun2.

Implementación del módulo de middle end con GIMPLE

En la etapa de GIMPLE el compilador realiza diversas tranformaciones del flujo del programa, ya sea para inlining, propagación de constantes, eliminación de redundancia, etc. Para ello el compilador ofrece una serie de estructuras de datos y funciones que permiten realizar tranformaciones del programa con relativa facilidad ya que son usadas para llevar a cabo optimizaciones como inlining, eliminación de redundancia, propagación de constantes, etc. Por otra parte los plugins del compilador también tienen acceso a estas funciones y tipos de datos, por lo que posibilita poder modificar el flujo del programa con relativa facilidad en un plugin. Esta es la principal razón por la que se eligió trabajar en GIMPLE y no en GENERIC por ejemplo, ya que si se buscara implementar el módulo de middle-end en GENERIC, básicamente habría que volver a implementar ciertas cosas que ya están dadas por el compilador para trabajar con GIMPLE, por ejemplo las funciones para reemplazar elementos en un objeto de tipo GIMPLE_CALL. Esto no solo implica más tiempo de trabajo y modificar código del compilador, sino que además hace más probable la introducción de bugs en el compilador y por esa razón es muy poco probable que un parche de esta naturaleza sea aceptado.

3. Ejemplo

En esta sección daremos un ejemplo de la utilización de la herramienta desarrollada.

Como primer paso, y por única vez en el sistema, debemos construir `tmock.so` usando los archivos `tmock.cpp` y `parser.cpp`:

```
> g++ -fno-rtti -I/[HEADERS] -c tmock.cpp -fPIC;g++ \
  -fno-rtti -I/[HEADERS] -c parser.cpp;g++ -fno-rtti \
  -I/[HEADERS] -c gimplePass.cpp
> g++ -fno-rtti tmock.o parser.o gimplePass.o -o tmock.so -shared
```

`tmock.so` es una librería dinámica que será linkeada cuando compilemos el test unitario. Con `tmock.so` y la lista de funciones mock (`list.mock`, ver más abajo), el compilador crea `tmock.h`, el cual deberá ser incluido por el test unitario para realizar el mocking de funciones o módulos.

Dados archivos a ser testeados, por ejemplo, `a.c` y `a.h` deseamos hacer una prueba unitaria con el uso de mocking. Los archivos son:

```
<currDir>/a.c:
~~~~~~~~~~~~~~
#include <stdio.h>
int functionExample(int num)
{
    return 0;
}
void caller(void)
{
    printf("%d\n", functionExample(5));
}

<currDir>/a.h:
~~~~~~~~~~~~~~
void caller(void);
```

y deseamos hacer mocking de la función `functionExample(int num)`.

Los pasos a seguir son los siguientes.

(1) Crear la lista de funciones a simular (`list.tmock`).

El formato del archivo es o bien un preámbulo seguido de líneas con funciones. El preámbulo puede ser utilizado, por ejemplo, para agregar includes necesarios por el prototipo de la las funciones de las que se hará mock (por ej. `#include <string.h>`, etc.). Las líneas del preámbulo deberán estar al comienzo del archivo y comenzar con el caracter ';'.

Las funciones a las que queremos hacer mock deben ser especificadas una por linea y el formato es

`lineType [expr]`

donde `lineType` puede ser sólo `1` o `2`.

Caso `lineType = 1`. En este caso `[expr]` será:

`identifier returnType functionName [listArgs]`

donde `functionName` será simulada en la función con `lineType = 2` y el mismo `identifier`. Si `identifier = -1`, entonces `functionName` será simulada en toda función que la utilice.

`[listArgs]` será una lista del tipo: `type1 type2 ... typeN`. Si la variable es de tipo const, se debe poner el prefijo `@` y si la variable es de tipo volatile se debe colocar a la variable el prefijo #.

Caso `lineType = 2`. En este caso `[expr]` será:

`identifier functionCaller`

donde `functionCaller`, es la función donde se usará la función simulada de la línea con `lineType = 1` y el mismo `identifier`.

En el ejemplo, el archivo `list.tmock`:

```
<currDir>/list.tmock:
~~~~~~~~~~~~~~~~~~~~~

1 7 int functionExample int
2 7 caller
```

En este archivo indicamos que `functionExample` será simulada ('mocked') en la función `caller`.
En este ejemplo es lo mismo si el archivo `list.tmock` es de la forma:

```
<currDir>/list.tmock:
~~~~~~~~~~~~~~~~~~~~~

1 -1 int functionExample int
```

Esto se debe a que en este caso estamos indicando que se simule la función `functionExample` en cada función que ocurra, pero la única función que la llama es `caller`.

(2) Incluir el archivo autogenerado `tmock.h`.

El archivo autogenerado `tmock.h` debe ser incluido en el archivo de test. Esto permitirá el usos de los siguientes macros:

- `*tmock_set_mock(functionExample, newFunction)` para cambiar la función `functionExample` por `newFunction` donde corresponda.
- `*tmock_restore_mock(functionExample)` para restablecer la función.

Este archivo incluye además las declaraciones adecuadas para las los wrapper de funciones y punteros de funciones. El archivo se genera automáticamente por el plugin en tiempo de compilación.

(3) Construir el archivo `test.c`.

Recordemos que en `a.c` la función `caller` imprime el valor que devuelve `functionExample(5)`.

Ahora definimos el archivo:

```
<currDir>/test.c:
~~~~~~~~~~~~~~~~~~~~~

#include "tmock.h" // se debe incluir el archivo autogenerado.
int functionExample_version1(int a)
{
    return a;
}

int main(void)
{
    caller();
    // Aquí se imprimirá 0 debido a que 'caller' llamará a
    // 'functionExample'.
    tmock_set_mock(functionExample, functionExample_version1);
    // Aquí asignamos 'functionExample_pointer' a
    // 'functionExample_version1'
    caller();
```

```
    // Aquí se imprimirá 5 debido a que  'caller' llamará a
    // 'functionExample_version1'.
    tmock_restore_mock(functionExample);
    // Here we are assingning 'functionExample_pointer'
    // back to 'functionExample'.
    caller();
    // Here 0 will be printed because caller will call
    // 'functionExample'.
    return 0;
}
```

En este caso `tmock.h` será:

```
<currDir>/tmock.h :
~~~~~~~~~~~~~~~~~~
#ifndef TMOCK_H
#define TMOCK_H
#define tmock_set_mock(function, new_function) \
function##_pointer = new_function
#define tmock_restore_mock(function) \
function##_pointer = function
int functionExample(int x0);
int(* functionExample_pointer)(int x0)=functionExample;
#endif
```

(4) Testing.

El ejemplo puede ser testeado de la siguiente manera:

```
>gcc -fplugin=<pluginDir>/tmock.so a.c test.c
```

En ejemplo, después que los plugins transforman el código, el código (interno) es equivalente a:

```
<currDir>/a.c(Transformed):
~~~~~~~~~~~~~~~~~~~~~~~~~~~
#include <stdio.h>
extern int(* functionExample_pointer)(int x0);

static inline int __attribute__ ((used)) functionExample_wrapper(int x0)
{
```

```
    return functionExample_pointer( x0);
}

// Estas líneas son inyectadas por el plugin en la etapa
// front-end del compilador

int functionExample(int)
{
    return 0;
}

void caller(void)
{
    printf("%d\n", functionExample_wrapper(5));

}
```

En la etapa middle-end del compilador la llamada a `functionExample` es reemplazada por `functionExample_wrapper`. Luego, modificando `functionExample_pointer`, podemos elegir si `functionExample_wrapper` será llamada como `functionExample` o como `functionExample_version1`. Por defecto, el valor de `functionExample_pointer` es `functionExample`. Esto está especificado en `tmock.h` que es generado por el plugin después de parsear `list.tmock`.

En la figura 4 se muestra resumidamente como es la secuencia de pasos a seguir en este ejemplo. También sirve para clarificar que pasos deben ser llevados a cabo por el usuario y cuales no.

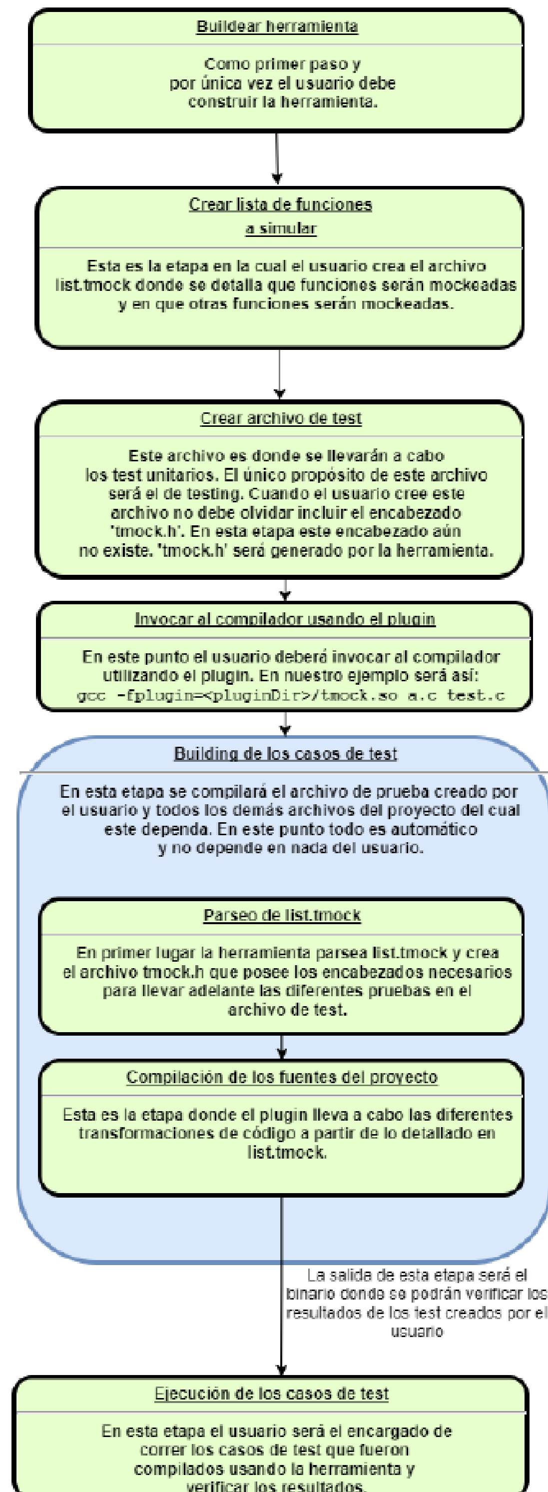


Figura 4. Pipeline del uso de la herramienta.

CAPíTULO 3

Conclusión

En este trabajo se logró crear una herramienta de mocking implementada como un plugin de GCC. Este plugin se encarga de realizar las tranformaciones de código necesarias en el front y middle-end del proceso de compilación. Esto permite la no modificación por parte del usuario del código a testear y que sea también plataforma independiente ya que estas etapas de compilación son independientes del sistema operativo y de la arquitectura del sistema (el sistema operativo y la arquitectura del sistema toman relevancia en la etapa de back-end).

El impacto en performance es mínimo ya que las tranformaciones de código llevadas a cabo por la herramienta solo requieren una indirección a la hora de llamar a las funciones que desean ser simuladas.

La implementación de la herramienta desarrollada requirió algunas modificaciones en el compilador las cuales fueron aceptadas y forman parte de GCC a partir de la versión 6.0.0. El parche con las modificaciones requeridas, cuyo autor es Andrés Tiraboschi, puede ser encontrado en el siguiente link: http://permalink.gmane.org/gmane.comp.gcc.patches/342255.

1. Trabajo futuro

De este trabajo se desprenden las siguientes continuaciones o mejoras:

- Ampliar el dominio de aplicabilidad a sistemas embebidos implementados en C++,
- realizar mocking de los llamados a punteros a funciones que apunten a una función a "mockear".

En el último punto una de las situaciones más problemáticas se da cuando por ejemplo se desea realizar mocking de una determinada de una función a la cual llamaremos `functionToMock`, solo en una determinada función que llamaremos `testFunction1`.

Luego tendremos un puntero a función que se llamará `funcPtr`, el cual es inicializado con `functionToMock` en el scope global. Este puntero luego será invocado en el cuerpo de `testFunction1` y en el de otra función a la cual no

deseamos modificar su comportamiento que llamaremos `testFunction2`. El código del ejemplo es el siguiente:

```
#include <stdio.h>

void functionToMock(void)
{
printf("Print example\n");
}

void (*funcPtr)(void) = functionToMock;

void testFunction1(void)
{
funcPtr();
}

void testFunction2(void)
{
funcPtr();
}
```

En este ejemplo si reemplazáramos la ocurrencia de `functionToMock` en la linea donde `funcPtr` es inicializada, estaríamos también modificando el comportamiento de `testFunction2` y eso no debería suceder. También debe tenerse en cuenta que los valores de los punteros a funciones podrían llegar a cambiar en runtime complicando aún más las cosas.

Bibliografía

[ED] *Making Unit Testing Practical for Embedded Development*, Electronic Design, [Online].

[MW] S. Mostafa y X. Wang, *An Empirical Study on the Usage of Mocking Frameworks in Software Testing*, Proceedings - International Conference on Quality Software, IEEE Computer Society, 127-132 (2014).

[MFC] T. Mackinnon, S. Freeman y P. Craig, *Endo-testing: unit testing with mock objects*, XP eXamined, Addison-Wesley, (2000).

[KWBF] M. Karlesky, G. Williams, W. Bereza y M. Fletcher, *Mocking the embedded world: Test-driven development*, Methods & Tools, Martinig & Associates (2007).

[K] S. S. Kim, *Mocking Embedded Hardware for Software Validation*, Tesis de Maestría del "Master of Science in Engineering", de la Universidad de Texas Austin (2016).

[T] T. Tromey, *Writing a GCC Front End*, Linux Journal **133**, http://www.linuxjournal.com/article/7884 (2005).