嵌入式軟體結構分析與測試
Structural Analysis and Testing of Embedded Software

劉建宏　　陳淑玲*　　王彥仁　　陳威諭

國立台北科技大學資訊工程系　　南台科技大學管理與資訊系*

Email: {cliu, t5598018, t7598006}@ntut.edu.tw　slchen@mail.stut.edu.tw*

摘要

近幾年嵌入式軟體成長迅速，被視為電子資訊產業提高其硬體產品競爭力的關鍵因素之一。然而嵌入式軟體往往與硬體和作業系統有密切的關係，並且缺乏結構化，使得嵌入式軟體較一般軟體複雜難懂，造成嵌入式軟體分析和測試的困難，測試案例的推導不易系統化或測試覆蓋率(coverage)不足，而無法提昇嵌入式軟體的品質與可靠性。因此本論文提出一個測試模型(test model)支援嵌入式軟體的結構分析和測試。此測試模型主要是擷取嵌入式軟體與硬體的介面資訊與程式的控制結構，協助測試人員了解嵌入式軟體的結構和推導測試案例。此外，我們亦開發了一個測試工具，可以自動建構所提出的測試模型，協助測試的進行，以減輕嵌入式軟體測試的負擔。

關鍵詞—嵌入式軟體、軟體測試、結構化測試

Abstract

In recent years, the number of embedded software has grown rapidly because embedded software has been considered as the key factor to enhance the competitiveness of products in information technology industry. However, embedded software is tightly coupled with hardware and operating system kernels and often lacks of structure, which makes embedded software complicated and difficult to understand, analyze, and test. As a result, it is extremely difficult to derive test cases systematically or to provide adequate test coverage for embedded software so as to improve its quality and reliability. In this paper, we propose a test model to support structural analysis and testing for embedded software. Specifically, the test model captures the software/hardware interface artifacts and control structures of embedded program. It can facilitate the understanding of embedded software and the derivation of test cases. In addition, a tool is developed to automate the construction of proposed model and to support test execution in order to reduce the effort required for testing embedded software.

Keyword—Embedded software, Software testing, Structural testing

一、前言

近年來，嵌入式系統逐漸成為我們生活中不可或缺的一部分，在手機、家電或一般 3C 產品裡都可以見到。隨著科技的進步，嵌入式系統的功能越來越強大，使得嵌入式軟體規模和複雜度亦不斷提高，造成嵌入式軟體的開發成本佔了整個嵌入式系統很大的比例。儘管嵌入式軟體重要性愈增，但嵌入式軟體的可靠性卻未隨之提高。特別是在某些特定環境，像是航空、汽車的電子系統中，嵌入式軟體的穩定和可靠度更顯為額外重要，倘若軟體發生錯誤，將會造成極大的損失。另外，嵌入式系統產品的開發，常因上市時間的壓力(time-to-market)，壓縮軟體開發的時程，以致於測試時間減少，而影響軟體的品質[2]，故需採用有效的測試方法來提昇嵌入式軟體的品質。

然而嵌入式軟體具有與一般軟體明顯不同的特性，例如：軟體與硬體有較高的耦合性、軟體開發環境和執行環境不同、多樣化的硬體執行平台與軟體開發工具、需符合硬體平台資源與時間的限制、缺乏清楚的設計模型等，造成嵌入式軟體的測試較為複雜。加上受限於作業系統與硬體平台資源的限制，一般軟體測試工具無法直接安裝或載入至嵌入式系統裝置來協助測試的進行，更使得嵌入式軟體的測試成為困難的挑戰。

由於嵌入式軟體往往與硬體和作業系統有密切的關係，並且缺乏結構化，使得嵌入式軟體複雜難懂，造成嵌入式軟體不易分析和測試，而無法提昇嵌入式軟體的品質與可靠性。

在這篇論文中，我們提出一個測試模型(test model)支援嵌入式軟體的結構分析和測試。此測試模型主要是擷取嵌入式軟體與硬體的介面資訊，例如：暫存器的初始化或記憶體的存取等，以及嵌入式軟體與作業系統的介面資訊，例如：作業系統程式核心函式的呼叫等，並將這些資訊描繪註解在嵌入式軟體程式的呼叫圖(call graph)、硬體介面關係圖和控制流程圖(control flow graph, CFG)上。此測試模型不但可有助於測試人員分析嵌入式軟體的結構，並可協助其找出嵌入式軟體中與硬體和作業系統有關的介面資訊與測試路徑。透過此測試模型，測試人員可以系統化地推導測試案例以驗證嵌入式軟體的結構和提升其結構的測試覆蓋率，進而達到提高嵌入式軟體品質的目的。

此外，我們並開發一個嵌入式軟體分析與測試的輔助工具 ESWAT(Embedded Software Analysis and Testing Tool)。此工具可以協助分析嵌入式軟體的原始程式碼，自動化建構所提出的測試模型，及推導測試路徑，以協助嵌入式軟體測試的進行。

本論文的組織架構如下：第二節為近年來與本研究相關的文獻探討與回顧，第三節透過範例說明所建立的嵌入式結構模型，第四節說明透過所建立的模型，如何協助推導測試案例，第五節則簡介本文所提出的嵌入式軟體分析與測試工具 ESWAT，第六節為本論文的結論與未來研究方向。

二、相關研究

隨著嵌入式系統的普及，以及嵌入式軟體規模與複雜度的增加，嵌入式軟體的品質亦逐漸受到重視，然而目前大多數的研究主要著重於嵌入式軟體的設計與開發，關於嵌入式軟體測試的文獻探討仍不多見，以下針對與本論文相關的研究做簡要的回顧。

Jerraya 和 Wolf [4]提出了一個硬體和軟體的共同設計方法。此方法在嵌入式系統的架構定義出抽象的硬體和軟體介面，包括軟體的 API 和硬體抽象層介面。接著進行軟體、硬體、與硬體有關(HW-dependent)的軟體以及與軟體有關(SW-dependent)的硬體進行共同設計，最後說明進行軟硬體共同驗證與軟硬體整合與測試的方法。

Lettnin 等人[5]提出一個基於覆蓋率(coverage)驅動的驗證方法以提高嵌入式軟體其變數和函式的覆蓋率。作者在開發主機上利用 SystemC 模擬 PowerPC 架構的嵌入式系統環境和驗證環境，並修改待測的嵌入式軟體程式碼，插入測試函式介面，作為驗證環境與模擬環境的介面，讓嵌入式軟體可以與模擬的硬體環境進行溝通，並協助產生嵌入式軟體在模擬環境上執行所需要的資料，及監視程式執行時的狀態。最後產生和執行測試案例，同時監控及驗證程式是否有缺陷。

Ma 和 Lim[6]針對嵌入式作業系統的驅動程式提出一個測試方法，作者認為驅動程式通常與嵌入式平台高度相依，但是相同類型的驅動程式則可以設計出一組共用的測試案例。因此提出能自動化產生可重複使用之測試案例的技術，該技術根據三種輸入資訊：(1)共用測試案例樣板；(2)與裝置驅動程式相依之測試資訊；(3)與系統平台相依之測試資訊，以自動化產生測試資料。

Seo 等人[8]提出一個自動化測試嵌入式軟體的架構 EmITM(Embedded System's Interface Test Model)，此架構植基於嵌入式硬體與作業系統介面的分析，以及模擬(emulation)嵌入式系統的環境。作者將程式中使用到硬體與作業系統介面的操作行為分類為 test feature，且根據每個 test feature 設計其測試案例樣板，最後根據這些樣板自動產生測試案例，並透過模擬測試技術(emulation test technique)執行已產生的測試案例。此技術是結合除錯與監控技術，在介面中插入中斷點(breakpoint)，驗證測試案例的結果為成功或失敗。

Seo 等人另外在[9]中指出嵌入式軟體測試需注意的一些問題：(1)嵌入式系統為高耦合性的軟體因此其整合測試非常重要；(2)介面越多的程式其錯誤可能越多；(3)在嵌入式軟體的錯誤中，與介面相關之錯誤比率。作者並且透過一個實際的開發案例來說明測試嵌入式軟體介面的重要性。

此外，Seo 等人在[10]中介紹一個用來支援介面測試技術的自動化測試工具—Justitia。此工

具是協助嵌入式軟體在測試週期的階段，對軟體進行整合測試，並為此工具提出在不同層級下軟體與軟體(software-software)和軟體與硬體(software-hardware)介面應成為一個新的覆蓋率準則(coverage criteria)，以及一個核心檢查點(core check-point)，幫助測試人員在測試執行時識別測試內容、選擇測試案例、以及分析測試的結果。

Sung 和 Choi[11] 提出一個測試資料選擇技術(test data selection technique)，可以發現嵌入式系統軟體與硬體之間互動的錯誤，此技術主要是根據嵌入式系統的需求規格產生對應的軟體程式來模擬該系統的行為，並將硬體可能產生的錯誤轉換成軟體錯誤，再透過錯誤注入技術(fault injection technique)[3]將這些錯誤注入至模擬程式中，最後選擇測試資料來偵測這些因軟體與硬體互動所引起的錯誤。作者並透過實際的案例來驗證此測試資料選擇技術的成效。

Tsai 等人[12]提出一個利用驗證模式(verification pattern)達到快速設計測試案例的方法。此方法是將系統相似的行為分類成劇本模式(scenario pattern)，對於每個劇本模式，測試工程師設計相對應的測試樣板(template)，用於測試具有相同行為的系統功能。如此，測試工程師可重複使用這些測試樣板來設計測試案例，以驗證相同行為模式的系統功能，並節省測試案例設計的時間與成本。

三、嵌入式軟體結構測試模型

在本節中，我們介紹所提出的嵌入式軟體結構測試模型，此模型結合嵌入式軟體的介面資訊至程式的呼叫圖、硬體介面關係圖與控制流程圖，協助測試人員了解嵌入式軟體的結構與介面，以推導測試案例。

(一) 嵌入式軟體的介面分析

嵌入式系統的平台架構通常包含應用程式(Application)、嵌入式作業系統(Embedded OS)、裝置驅動程式(Device Driver)、韌體(Firmware)和硬體(Hardware)所組成，如圖 1 所示。一般而言，應用程式為不包含作業系統或硬體介面之程式，例如純運算之程式；作業系統則負責管理電腦軟體與硬體的資源，例如 linux 與 uClinux 等；裝置驅動程式為 LCD、鍵盤或 USB 裝置的驅動程式；韌體為初始化硬體裝置或負責中斷處理與 Boot loader。

由於嵌入式軟體主要是儲存在嵌入式系統的平台上，並透過作業系統或驅動程式與硬體互動來完成所需要的功能。因此，在本論文中，我們將嵌入式軟體與外部軟硬體的介面，分成三種，分別為與應用程式相關的介面、與作業系統相關的介面、以及與硬體相關的介面。其中應用程式介面為應用程式與應用程式之間的介面，與作業系統和硬體無關，例如測試人員自訂函式之間的呼叫；作業系統介面為應用程式與作業系統之間的介面，例如應用程式對作業系統函式的呼叫；硬體相關介面則為應用程式與硬體相關的介面，例如應用程式透過自行定義的變數對暫存器或記憶體等硬體裝置進行操作。以下我們將針對作業系統介面與硬體相關介面進一步說明。

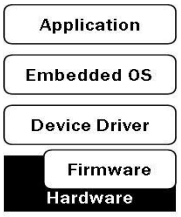


圖 1、嵌入式系統的平台架構

(1) 作業系統介面(OS interface)

作業系統介面為嵌入式軟體程式中與作業系統相關的部份，通常為對作業系統核心(kernel)所提供之函式的呼叫。圖 2 為嵌入式軟體程式中具有作業系統介面的範例。

```
1 //OS interface example
2 static void __exit cs8900_cleanup (void) {
3     int ret;
4     cs8900_t *priv = (cs8900_t *) cs8900_dev.priv;
5     if ( priv->char_devnum) {
6         ret = unregister_chrdev(priv->char_devnum,"cs8900_eeprom");
7     }
8     release_mem_region(cs8900_dev.base_addr,16);
9     unregister_netdev(&cs8900_dev);
10 }
```

圖 2、作業系統介面範例程式

在圖 2 範例程式中，第 6、8 和 9 行使用到作業系統核心函式 unregister_chrdev()、

release_mem_region()與 unregister_netdev()，因此，範例程式的第 6、8 和 9 行存在與作業系統相關的介面。

作業系統介面的存在代表嵌入式軟體程式與作業系統具有相依關係，當作業系統有變更時，如 linux kernel 的更新等或作業系統的轉換，嵌入式軟體程式需要針對其與作業系統相關的介面進行測試，例如測試參數傳遞的形態與返回的資料型態是否正確，以確保作業系統修改後該介面的正確性。

(2) 硬體介面(Hardware interface)

硬體介面為與嵌入式軟體程式中與硬體相關之介面，例如暫存器的設定、記憶體配置或各種 I/O 的使用。硬體介面通常與開發平台硬體有很大的相依性，因此在進行平台的轉換(porting)或更換時，嵌入式軟體程式必須針對其與硬體介面相關的部份進行測試，以確保平台更新後系統的正確性。

另外，為使程式容易理解與維護，開發人員通常會在嵌入式軟體或裝置驅動程式中，利用 C 語言的標頭檔來描述其硬體的位址與內容。利用容易識別的變數代替硬體位址，除了可改善程式的可讀性外，並可讓程式較容易進行移植。倘若硬體變更導致記憶體位址配置改變時，程式僅需修改其標頭檔中相關的敘述即可。因此，可透過標頭檔的分析來協助辨識嵌入式軟體的硬體介面。

圖 3 為存在硬體介面的程式範例。此程式在第五行設定變數 NFCONF 代表 NAND Flash Configuration Register 暫存器的位址 0xf830。當程式使用 NFCONF 變數進行運算時，即代表對記憶體位址 0xf830 進行操作。因此，範例程式的第 5 行存在與硬體相關的介面。另外，圖 3 範例程式在第六行呼叫開發人員自定的函式 reset_nand()，所以第 6 行存在一個與應用程式相關的介面。

```
1 //HW interface example
2 //initialized NAND Flash
3
4 void init_nand (void) {
5     NFCONF = 0xf830;
6     reset_nand();
7 }
```

圖 3、硬體界面範例程式

(二) 嵌入式軟體結構測試模型

嵌入式軟體結構測試模型主要包含呼叫圖、硬體介面關係圖與控制流程圖。其中呼叫圖和硬體介面關係圖用來顯示個別函式其軟硬體的介面資訊，可協助關於介面測試資料的推導。控制流程圖則用來顯示函式的控制結構，可用來協助推導測試路徑。藉由將作業系統與硬體的介面資訊註記在函式的控制流程圖上，可以協助設計測試案例來檢驗嵌入式軟體的控制流程結構，同時可以測試其與軟硬體介面之間的正確性。

為說明嵌入式軟體測試結構模型，本節將利用 linux kernel 2.6.11 版本之 CS8900A 網路晶片驅動程式中的函式 cs8900_receive()為例，如圖 4 所示，介紹其對應的呼叫圖、硬體介面關係圖與控制流程圖。

```
2 static void cs8900_receive (struct net_device *dev) {
3         cs8900_t *priv = (cs8900_t *) dev->priv;
4         struct sk_buff *skb;
5         u16 status,length;
6
7         status = cs8900_read (dev,PP_RxStatus);
8         length = cs8900_read (dev,PP_RxLength);
9
10        if (!(status & RxOK)) {
11                priv->stats.rx_errors++;
12                if ((status & (Runt | Extradata)))
13                        priv->stats.rx_length_errors++;
14                if ((status & CRCerror))
15                        priv->stats.rx_crc_errors++;
16                return;
17        }
18
19        if ((skb = dev_alloc_skb (length + 4)) == NULL &&
20                (length+ 4 ) > MAX_LENGTH ) {
21                priv->stats.rx_dropped++;
22                return;
23        }
24
25        skb->dev = dev;
26        skb_reserve (skb,2);
27        cs8900_frame_read (dev,skb,length);
28        skb->protocol = eth_type_trans (skb,dev);
29        netif_rx (skb);
30
31        dev->last_rx = jiffies;
32        priv->stats.rx_packets++;
33        priv->stats.rx_bytes += length;
34 }
35
36 static inline u16 cs8900_read (struct net_device *dev,u16 reg) {
37        outw (reg,dev->base_addr + PP_Address);
38        return (inw (dev->base_addr + PP_Data));
39 }
40
41 static inline void cs8900_frame_read (struct net_device *dev,
42                                       struct sk_buff *skb,u16 length) {
43        insw (dev->base_addr,skb_put (skb,length),(length + 1) / 2);
44 }
```

AP

OS

圖 4、cs8900_receive()範例程式

(1) 呼叫圖與硬體介面關係圖

呼叫圖可以顯示程式中函式之間的呼叫關係，並提供作業系統與應用程式介面的相關資訊，讓測試人員可以瞭解程式的呼叫結構和函式之間彼此的相依關係，這些資訊有助於嵌

入式軟體整合測試(integration testing)或迴歸測試(regression testing)的進行。

圖 5 為 cs8900_receive()的呼叫圖，該圖顯示 cs8900_receive()與 6 個函式有呼叫關係，其中函式 dev_alloc_skb()、skb_reserve()、netif_rx()與 eth_type_trans()屬於作業系統的函式呼叫，而函式 cs8900_read()與 cs8900_frame_read()則屬於應用程式的函式呼叫。圖 5 並顯示作業系統與應用程式介面的相關資訊，包含作業系統函式的標頭檔、函式呼叫的參數和型態、以及函式返回的資料型態等。

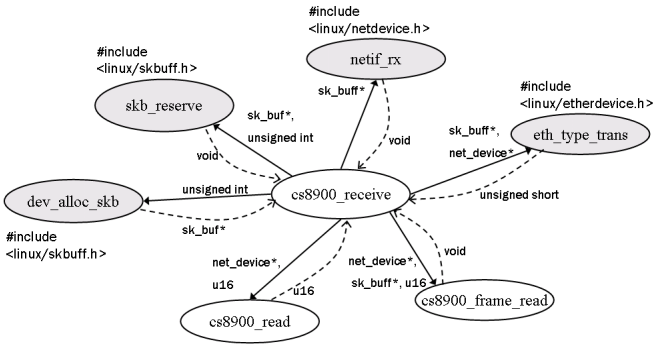


圖 5、cs8900_receive()的呼叫圖

另外，硬體介面關係圖是用來顯示函式其硬體介面的資訊，讓測試人員可以瞭解函式中哪些變數代表硬體的暫存器或記憶體，以及變數內的資料是被寫入至硬體或是從硬體中被讀取。此外，硬體介面關係圖並可標示這些暫存器或記憶體的相關資料，例如位元大小，讓測試人員能夠根據這些資訊設計與硬體介面相關的測試案例。

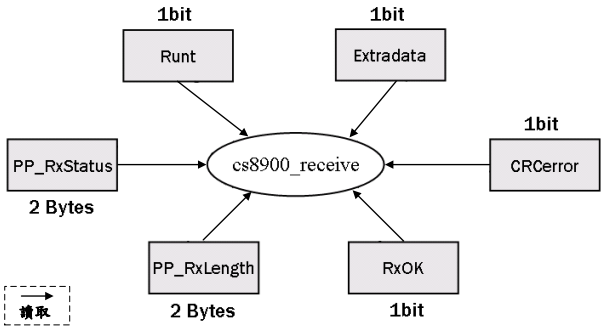


圖 6、cs8900_receive()硬體介面關係圖

圖 6 為 cs8900_receive()的硬體介面關係圖，代表 cs8900_receive()函式的哪些變數跟硬體有關，以及這些變數對應之硬體的位元大小。圖 6 顯示 cs8900_receive()有 6 個變數，Runt、Extradata、CRCerror、PP_RxStatus、PP_RxLength 和 RxOK。透過這些「硬體變數」的操作，cs8900_receive()可以設定(寫入)或使用(讀取)在對應之暫存器或記憶體內的資料。

(2) 控制流程圖

控制流程圖主要是用來表示程式的控制流程結構，並可從中推導程式可能的執行路徑。控制流程圖通常以節點(node)與邊(edge)所組成，其中節點代表程式的敘述或區塊(statement block)，邊則代表程式的執行順序(即控制流)。為了在控制流程圖中註記嵌入式軟體的介面資訊，本論文利用不同的節點符號來表示不同的介面組合，如作業系統、硬體、或應用程式等，如圖 7 所示。

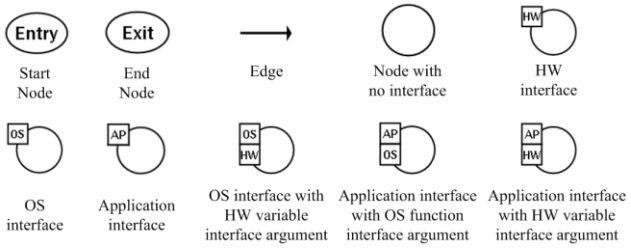


圖 7、控制流程圖組成元件

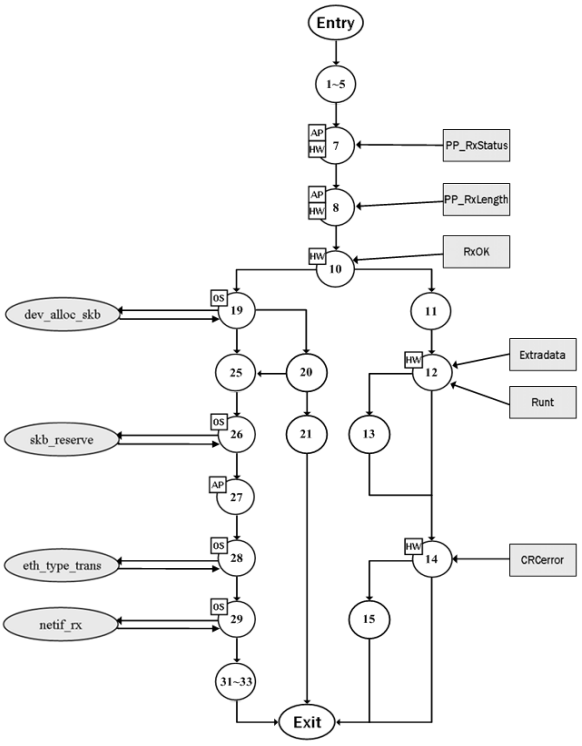


圖 8、結合介面資訊之控制流程圖

圖 8 為 cs8900_receive()其加註介面資訊的控制流程圖。透過這些介面資訊，可以辨識程式中哪些執行路徑與作業系統或硬體有關。當作業系統或硬體平台開發完成或變更時，即可針對嵌入式軟體中與其相關之介面，找出經過這些介面的路徑來進行測試。例如圖 8 中，節點 19 包含作業系統介面。因此當節點 19 相關的作業系統函式改變時，如作業系統程式庫版本更新，則經過節點 19 的相關路徑必需進行測試，以確保變動後程式的正確性。

四、測試案例的推導

第三節所提的測試模型不但有助於了解待測的嵌入式軟體程式，亦可協助測試案例的推導。圖 9 說明從建構測試模型至產生測試案例的流程。首先需在嵌入式軟體的程式碼內標示出與作業系統和硬體介面相關的變數或標頭檔等資訊，接著藉由分析程式碼產生呼叫圖、硬體介面關係圖與控制流程圖，並從圖形中推導出測試路徑，且根據各個路徑所包含的介面資訊和限制條件設計測試案例，並撰寫測試程式碼(Test Script)。然後依據測試覆蓋準則(coverage criteria)[7]選擇出適合的測試案例，並將測試案例載入目標測試板(target board)以執行測試和產生測試結果。以下將針對測試路徑推導和測試案例設計做詳細說明。

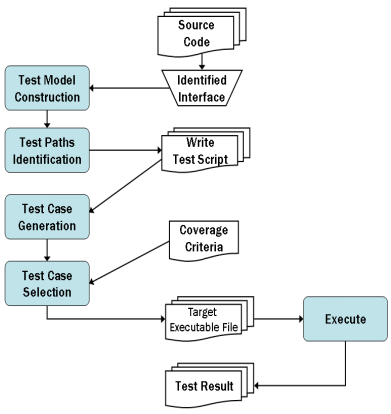


圖 9、測試流程圖

(一) 基本路徑(basis path)的推導

基本路徑測試(basis path testing)是白箱測試(white-box testing)技術的一種[7]，藉由測試函式的基本路徑(又稱獨立路徑)，可以確保函式中每一行程式均會被執行，亦即達到 100%敘述覆蓋率(statement coverage)[1]，以確保程式品質和增加程式正確性的信心。

基本路徑是以控制流程圖為基礎，透過分析控制流程圖的複雜度(cyclomatic complexity)[7]，可以得知基本路徑的數量，進而推導函式的基本路徑。一般而言，每條基本路徑是從控制流程圖的節點 Entry 開始至節點 Exit 結束，並至少會經過一個以上尚未被其他基本路徑涵蓋的邊(edge)[7]。以圖 4 範例程式而言，透過 cyclomatic complexity 的計算，可以推導出六條基本路徑，如表 1 所列。並可將路徑依照經過的介面進行分類如表 2，路徑共分為三類，分別為：(1)與作業系統介面相關路徑；(2)與硬體介面相關路徑；(3)與應用程式介面相關路徑。透過介面的分類可讓測試人員得知基本路徑與何種介面有關，並可根據硬體或作業系統等介面的資訊，協助設計通過這些基本路徑的測試案例。

表 1、cs8900_receive()之基本測試路徑

編號	路徑順序
P1	Entry − (1~5) − 7 − 8 − 10 − 19 − 25 − 26 − 27 − 28 − 29 − (31~33) − Exit
P2	Entry − (1~5) − 7 − 8 − 10 − 19 − 20 − 25 − 26 − 27 − 28 − 29 − (31~33) − Exit
P3	Entry − (1~5) − 7 − 8 − 10 − 19 − 20 − 21 − Exit
P4	Entry − (1~5) − 7 − 8 − 10 − 11 − 12 − 14 − 15 − Exit
P5	Entry − (1~5) − 7 − 8 − 10 − 11 − 12 − 13 − 14 − Exit
P6	Entry − (1~5) − 7 − 8 − 10 − 11 − 12 − 14 − Exit

表 2、cs8900_receive()之基本測試路徑與介面分類

相關介面路徑	路徑編號
與作業系統介面相關路徑	P1, P2, P3
與硬體介面相關路徑	P1, P2, P3, P4, P5, P6
與應用程式介面相關路徑	P1, P2

(二) 測試案例的設計

在得知基本路徑經過之介面相關資訊後，測試人員可以分析路徑上與硬體介面有關之硬體變數(或與作業系統介面有關之函式參數)的值域(domain)，並利用等價劃分(equivalence partition)或邊界值分析(boundary value analysis)

[1][7]等測試技術來設計測試案例，以驗證不同硬體變數值(或作業系統函式參數值)下的測試情境。例如，當硬體變數或函式參數之值域為布林值時(亦即硬體為 1 位元)，利用等價劃分的技術可分別對硬體變數值為 0 與 1 或函式參數值為真(true)與偽(false)的情況進行測試；而當硬體變數或函式參數之值域為區間範圍(range)時(亦即硬體為多位元組)，可利用等價劃分技術對區間內外，及利用邊界值分析技術對區間之邊界值分別設計對應的測試案例；而當值域為集合時(亦即硬體組態可切換為不同行為模式)，則利用等價劃分技術可對每一種情境設計出對應的測試案例。

以圖 4 cs8900_receive()為例，其基本路徑 P5(見表 1)經過圖 8 的節點 12，並於該節點使用到硬體變數 Runt 與 Extradata，透過圖 6 可得知變數 Runt 與 Extradata 皆為硬體變數，代表暫存器的 1 個位元(bit)，因此可利用等價劃分技術將變數 Runt 與 Extradata 的內容設定為 0 或 1，並可推導出 00、01、10、11 四種組合。不過其中 Runt=0 與 Extradata=0 組合會造成 P5 不會被執行 (此組合被路徑 P4 或 P6 涵蓋)，因此我們可以設計三個測試案例，如表 3 所列，以驗證不同的測試情境。

表 3、驗證 Runt 與 Extradata 硬體介面之測試案例

	輸入資料	預期結果
TC1	dev, cs8900_clear (dev, RxEvent, RxOK), cs8900_clear(dev, RxEvent, Runt), cs8900_set(dev, RxEvent, Extradata)	dev->priv->stats .rx_Length_ errors +1
TC2	dev, cs8900_clear (dev, RxEvent, RxOK), cs8900_set(dev, RxEvent, Runt), cs8900_clear(dev, RxEvent, Extradata)	dev->priv->stats .rx_Length_ errors +1
TC3	dev, cs8900_clear (dev, RxEvent, RxOK), cs8900_set(dev, RxEvent, Runt), cs8900_set(dev, RxEvent, Extradata)	dev->priv->stats .rx_Length_ errors +1

在表 3 中，測試案例 TC1、TC2 與 TC3 的輸入資料中皆使用代表裝置的變數 dev，以及函式 cs8900_set()與 cs8900_clear()，其中 dev 為 cs8900_receive()的輸入，cs8900_clear(dev, RxEvent, RxOK)會將裝置 dev 的 RxEvent 暫存器中 RxOK 欄位清除為 0，讓路徑 P5 可以被執行。而 cs8900_set(dev, RxEvent, Runt)和 cs8900_set(dev, RxEvent, Extradata)則分別將裝置 dev 的 RxEvent 暫存器中 Runt 和 Extradata 欄位設定為 1。同樣的，cs8900_clear(dev, RxEvent, Runt)和 cs8900_clear(dev, RxEvent, Extradata)則將裝置 dev 的 RxEvent 暫存器中 Runt 和 Extradata 欄位清除為 0。這三個測試案例的預期結果均會執行圖 4 cs8900_receive()中的第 13 行，亦即將輸入裝置 dev 之資料結構中的 rx_Length_error 變數加一。

另外，基本路徑 P1 經過節點 19，透過分析得知該節點包含作業系統介面 dev_alloc_skb()，且該介面有一個參數(length+4)，其中 length 的資料型態為 unsigned int，其值域範圍為 0~65535。因此，透過邊界值分析的技術，可針對函式 dev_alloc_skb()設計 length=0, length=32767, length=65535 的測試情境，如表 4 所列。

在表 4 中，測試案例 TC4、TC5 與 TC6 皆將暫存器 RxEvent 的 RxOK 欄位設定為 1，讓路徑 P1 可以被執行。另外，由於參數 length 的值可透過暫存器 PP_RxLength 來設定，故可將 PP_RxLength 的內容分別設定為 0、0x7FFF(即 $32767_{(10)}$)與 0xFFFF(即 $65535_{(10)}$)。這些測試案例的預期結果皆為接收成功，並使 dev 資料結構中的 rx_packet 加一。

表 4、驗證 dev_alloc_skb()作業系統介面之測試案例

	輸入資料與環境變數的設定	預期結果
TC4	dev, cs8900_set(dev, RxEvent, RxOK), cs8900_set(dev, PP_RxLength, 0x0)	dev->priv ->stats.rx _packets+1
TC5	dev, cs8900_set(dev, RxEvent, RxOK), cs8900_set(dev, PP_RxLength, 0x7FFF)	dev->priv ->stats.rx _packets+1
TC6	dev, cs8900_set(dev, RxEvent, RxOK), cs8900_set(dev, PP_RxLength, 0xFFFF)	dev->priv ->stats.rx _packets+1

五、嵌入式軟體測試工具

為支援所提出的結構測試模型，以及協助嵌入式軟體的測試，我們開發一個嵌入式軟體分析與測試的輔助工具 ESWAT。此工具主要目的在於解析嵌入式軟體結構，透過分析產生相關資訊，以供測試人員推導測試案例，並驗證其測試結果。

圖 10 為本工具之系統架構圖，主要分為本機端的 ESWAT 系統與目標端的 Test Harness 兩個部份。在本機端的 ESWAT 系統主要為負責程式的靜態分析，如產生呼叫圖、控制流程圖、以及程式基本路徑等，提供測試人員設計測試案例和撰寫測試程式碼的參考。此外，ESWAT 亦可支援將測試案例程式碼與 Test Harness 系統的程式進行連結與交叉編譯，產生可執行於目標端之執行檔。而在目標端的 Test Harness 為目標平台相依(target-dependent)之程式碼，主要負責目標板的初始化、連接、測試案例的執行與記錄其執行路徑等，以提供 ESWAT 系統進行測試結果記錄檔的分析。

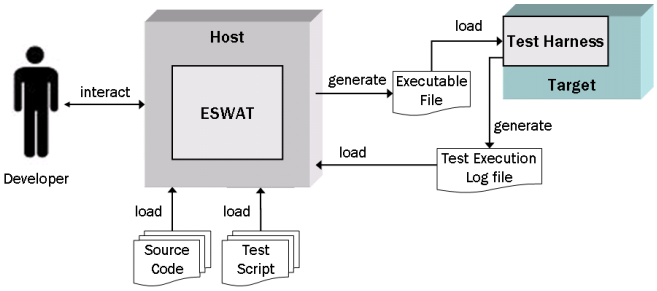
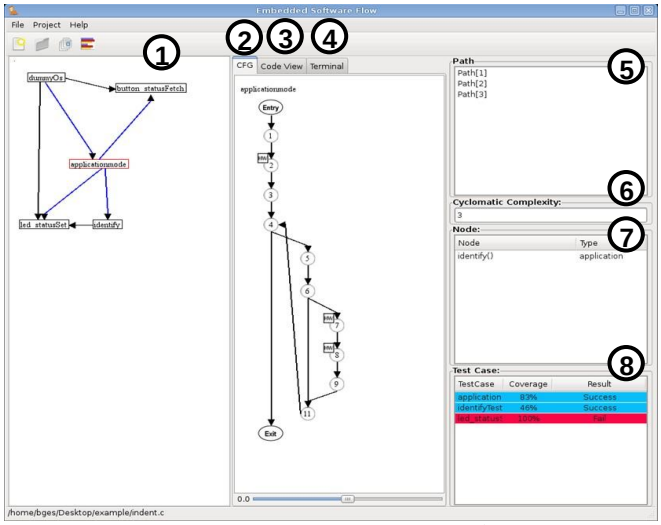


圖 10、ESWAT 系統架構圖



①呼叫圖 ②控制流程圖 ③程式碼顯示區 ④終端機畫面 ⑤基本路徑列表 ⑥複雜度計算 ⑦節點資訊 ⑧測試案例列表與測試相關資訊

圖 11、ESWAT 工具區塊說明

圖 11 為 ESWAT 測試工具執行畫面的範例。此畫面顯示待測嵌入式軟體程式的呼叫圖、控制流程圖、cyclomatic complexity 複雜度分析、基本路徑、路徑節點資訊、以及測試案例等資訊。基本上，測試人員將待測程式的原始碼匯入至 ESWAT 後，ESWAT 便針對原始碼中所有的函式進行分析，並產生和顯示呼叫圖(圖 11 區塊①)。測試人員可點選呼叫圖中任一個函式，以觀看該函式對應的控制流程圖(圖 11 區塊②)。ESWAT 會在控制流程圖中標示出與作業系統與硬體介面相關之節點，測試人員可點選這些節點查詢節點的相關資訊，包含該節點相關之函式、判斷式條件、以及介面之類型等(圖 11 區塊⑦)。

此外， ESWAT 將依據所選定之函式的控制流程圖，推導出其 cyclomatic complexity 複雜度與可能的基本路徑(圖 11 區塊⑤⑥)，測試人員可選擇任一條路徑，ESWAT 會在程式原始碼顯示區塊(Code View)反白顯示該路徑所涵蓋的程式敘述，例如圖 12 顯示路徑 Path[2]所涵蓋的程式敘述。

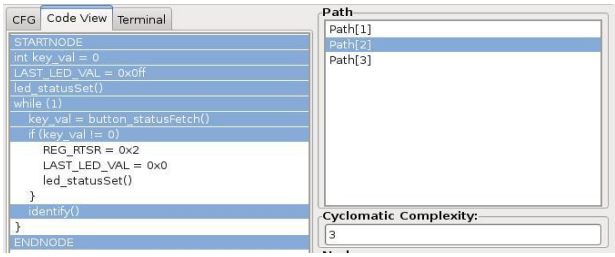


圖 12、可執行之基本路徑

根據 ESWAT 所列出的基本路徑，測試人員可以參酌執行該路徑的限制條件和相關軟硬體介面資訊，以設計適當的測試案例和撰寫測試程式碼。此測試程式碼將與嵌入式程式共同編譯，並載入目標板執行，其測試結果將於 ESWAT 顯示(圖 11 區塊⑧)，包含測試案例名稱、該測試案例的敘述涵蓋率、及測試案例是否通過(Success)或是失敗(Fail)等資訊。

六、結論與未來研究方向

本篇論文中我們提出一個嵌入式軟體的結構測試模型，包含函式呼叫圖、硬體介面關係圖、和控制流程圖，以萃取嵌入式軟體之應用程式、作業系統與硬體介面資訊。這些資訊不但有益於測試人員了解嵌入式軟體的結構，並可用來推導測試路徑。同時透過分析測試路徑的執行條件、相關之硬體變數或作業系統函式參數等資訊，可協助測試人員設計不同

的測試案例，以驗證各種可能的測試情境，進而提高程式結構的覆蓋率，確保嵌入式軟體的品質。此外，本論文亦開發一個嵌入式軟體分析與測試的輔助工具 ESWAT，可自動建構所提出的測試模型和協助測試的進行，以減輕嵌入式軟體測試的負擔。

對於未來的研究方向，我們將朝以下幾個項目持續發展：(1)深入探討硬體變數和作業系統函式介面對嵌入式軟體結構化測試的影響；(2)應用資料流(data flow)分析技術於嵌入式軟體的測試；(3)持續改善嵌入式軟體測試輔助工具 ESWAT；以及(4)應用硬體或作業系統介面相關資訊於嵌入式軟體的迴歸測試。

七、參考文獻

[1] Boris Beizer, *Software Testing Techniques*, 2nd Edition, Van Nostrand-Reinhold, 1990.

[2] Bart Broekman and Edwin Notenboom, *Testing Embedded Software*, Addison Wesley, 2003.

[3] Mei-Chen Hsueh, Timonthy K. Tsai, and Ravishankar K. Lyer, "Fault Injection Techniques and Tools," *IEEE Computer*, pp. 75-82, 1997.

[4] Ahmed A. Jerraya and Wayne Wolf, "Hardware/Software Interface Codesign for Embedded Systems," *Computer*, Vol. 38, Issue 2, pp. 63-69, 2005.

[5] Djones Lettnin, Markus Winterholer, and Axel Braun, "Coverage Driven Verification Applied to Embedded Software," *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, pp. 159-164, 2007.

[6] Yu-Seung Ma and ChaeDeok Lim, "Test System for Device Drivers of Embedded System," *International Conference of Advanced Communication Technology*, Vol. 1, pp. 550-552, 2006.

[7] Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, 5th Edition, McGraw-Hill, 2001.

[8] Jooyoung Seo, Ahyoung Sung, Byoungju Choi, and Sungbonk Kang, "Automating Embedded Software Testing on an Emulated Target Board," *Proceedings of the Second International Workshop on Automation of Software Testing*, pp. 9-15, 2007.

[9] Jooyoung Seo, Yuhoon Ki, Byoungju Choi, and Kwanghyun La, "Which Spot Should I Test for Effective Embedded Software Testing?," *Proceedings of the 2008 second International Conference on Secure System Integration and Reliability Improvement*, Vol. 0, pp. 135-142, 2008.

[10] Jooyoung Seo, Yuhoon Ki, Byoungju Choi, and Kwanghyun La, "Tool Support for New Test Criteria on Embedded Systems: Justitia," *Proceedings of the Second International Conference on Ubiquitous Information Management and Communication*, pp. 365-369, 2008.

[11] Ahyoung Sung and Byoungju Choi, "An Interaction Testing Technique between Hardware and Software in Embedded Systems," *Proceedings of the Ninth Asia-Pacific Software Engineering Conference*, pp. 457-464, 2002.

[12] Wei-Tek Tsai, Lian Yu, Feng Zhu, and Ray Paul, "Rapid Embedded System Testing Using Verification Patterns," *IEEE Software*, Vol. 22, Issue 4, pp. 68-75, July 2005.