ZESZYTY NAUKOWE POLITECHNIKI ŚLĄSKIEJ 2014
Seria: ORGANIZACJA I ZARZĄDZANIE z. 74 Nr kol. 1921

Sebastian BIEŃ, Ireneusz J. JÓŹWIAK
Wydział Informatyki i Zarządzania
Politechnika Wrocławska

LOGGING MANAGEMENT LANGUAGE – JĘZYK DO ZARZĄDZANIA INSTRUMENTACJĄ KODU ŹRÓDŁOWEGO

Streszczenie. W artykule opisano język do automatycznego zarządzania instrumentacją kodu źródłowego podczas testowania integracyjnego systemów wbudowanych, zwany językiem LML. Opisano składnię tego języka oraz elementy, z których się składa. W artykule zostały także zawarte opis i zastosowanie plików koniecznych do działania tego języka. Omówiono przykładową implementację procesora języka LML oraz przeprowadzono analizę przydatności języka LML.

Słowa kluczowe: język LML, instrumentacja kodu źródłowego, system wbudowany, testowanie integracyjne.

LOGGING MANAGEMENT LANGUAGE – LANGUAGE FOR MANAGEMENT OF SOURCE CODE INSTRUMENTATION

Summary. The paper describes the language for automated management of source code instrumentation during integration testing of embedded systems. It is described the syntax of the language and the elements that compose it. The paper also contains a descriptions and application files needed to run the language. It is shown an example implementation of the language procesor and LML conducted a brie discussion of suitability of language LML.

Keywords: LML language, source code instrumentation, embedded system, integration testing.

1. Wprowadzenie

Język LML (ang. *Logging Management Language*) jest prostym, deklaratywnym językiem służącym do zarządzania instrumentacją kodu źródłowego podczas testowania

integracyjnego oprogramowania wbudowanego, które zostało napisane w języku C. Ekspresja języka LML pozwala na skrócenie czasu potrzebnego na instrumentację kodu źródłowego oraz pozwala opisać komponenty składowe oprogramowania. Aby możliwe było dalsze opisywanie tego języka, należy wyjaśnić takie pojęcia, jak: testowanie integracyjne, systemy wbudowane oraz, co najważniejsze, czym jest instrumentacja kodu źródłowego [6].

Testy integracyjne, tak jak to definiuje inżynieria oprogramowania, mają na celu testowanie oprogramowania podczas jego scalania z części składowych. Niektórych błędów nie da się znaleźć w trakcie testowania samych jednostek składowych. Są to błędy związane z interfejsami pomiędzy komponentami. Podczas fazy testowania integracyjnego powinny zostać znalezione błędy związane z komunikacją pomiędzy komponentami. Rozróżniamy dwa rodzaje testów integracyjnych [7]:

- *modułowe* – umożliwiają sprawdzenie, czy dane pomiędzy modułami są przekazywane poprawnie,
- *systemowe* – umożliwiają sprawdzenie, czy dane pomiędzy systemami są przekazywane poprawnie.

System wbudowany (ang. *Embedded System*) [8] jest to system komputerowy specjalnego przeznaczenia, który staje się integralną częścią obsługiwanego przez niego sprzętu. System wbudowany spełnia określone wymagania, zdefiniowane do zadań, które ma wykonywać. Nie można nim więc nazywać typowego wielofunkcyjnego komputera osobistego. Każdy system wbudowany jest oparty na mikroprocesorze (lub mikrokontrolerze), zaprogramowanym do wykonywania ograniczonej liczby zadań lub nawet tylko do jednego.

Instrumentacja kodu źródłowego odnosi się do zdolności monitorowania lub pomiaru wydajności kodu w celu zdiagnozowania błędów. James H. Andrews w pracach [1, 2] przedstawił koncepcję zarządzania instrumentacją kodu. W wyniku analizy uzyskuje odpowiedź, czy zastosowany test przechodzi, czy nie. Andrews zaproponował użycie specjalnego języka opisującego analizę. W Polsce wybitne zasługi w rozwoju zarządzania instrumentacją kodu źródłowego ma Dominik Hura [3, 4, 5]. Programiści implementują instrumentację w postaci instrukcji kodu, monitorujących określone komponenty systemu. Jeśli aplikacja zawiera instrumentację kodu, to może być zarządzana przy użyciu specjalistycznych narzędzi do tego służących. Instrumentacja jest niezbędna do pomiaru wydajności aplikacji. Wyróżniamy następujące podejścia do instrumentacji: instrumentację kodu źródłowego oraz instrumentację binarną.

W niniejszym artykule podjęto próbę zaproponowania języka do zarządzania instrumentacją kodu źródłowego.

2. Zarządzanie instrumentacją kodu źródłowego

Działające na mikrokontrolerach oprogramowanie czasu rzeczywistego najczęściej jest pisane w języku C. W innych językach, np. obiektowych (Java, C#), pisane są tylko warstwy związane z interfejsem. Oprogramowanie to często do swojego działania wykorzystuje systemy operacyjne czasu rzeczywistego, gdyż umożliwiają one komunikację przez przesyłanie komunikatów. Pomimo to programiści nadal jednak używają interfejsów funkcyjnych jako podstawowego środka komunikacji między komponentami.

Najpopularniejszą metodą służącą do otrzymywania danych o zachowaniu opracowanego oprogramowania jest logowanie (zapisywanie do logów). Polega ono na rejestrowaniu przebiegu działania oprogramowania w plikach dziennika, które są przechowywane w bazach danych lub pamięci masowej. Bywa, że jest to jedyna metoda, dzięki której możemy poznać zachowanie programu „od środka". Jednak rejestrowana informacja może być bardzo szybko wyprowadzona poza urządzenie dzięki dostępnym w nim interfejsom sprzętowym. Do tego stosuje się właśnie instrumentację kodu źródłowego. Polega ona na modyfikacji istniejącego kodu przez dodanie do niego dodatkowego kodu lub przez transformację istniejących już fragmentów.

Komunikacja pomiędzy komponentami w systemach wbudowanych czasu rzeczywistego jest zazwyczaj implementowana przy użyciu interfejsów funkcyjnych i mechanizmów przesyłania komunikatów. Innym sposobem jest użycie zmiennych dzielonych. Jednak jest to niezalecane ze względu na wprowadzanie do oprogramowania niepożądanych zależności między modułami. Zatem należy rejestrować wywołania funkcji interfejsowych poszczególnych komponentów oraz ruch w systemie podczas konkretnych sekwencji komunikacyjnych. Każdy komunikat zarówno przy wysłaniu, jak i odbiorze powinien być przechwycony w celu analizy potrzebnych informacji. Najistotniejszymi informacjami, które powinny być wydobyte z wywołań funkcji podczas testowania integracyjnego, są:

1. nazwa wywołującego,
2. czas wystąpienia funkcji w urządzeniu,
3. wartość zwrócona z funkcji,
4. wartości parametrów wejściowych do funkcji,
5. wartości parametrów wyjściowych z funkcji,
6. nazwy typów parametrów wej/wyj i wartości zwróconej z funkcji.

Informacje, które powinny być wydobyte przez przesyłanie komunikatów, są następujące:

1. identyfikator nadawcy,
2. identyfikator odbiorcy,
3. identyfikator komunikatu,
4. dane przesyłane wraz z komunikatem,
5. czasy wysłania i odebrania komunikatu.

Informacje te są przydatne w analizie sekwencji komunikacyjnych między komponentami. Taka analiza pozwala na porównanie parametrów funkcji i wartości zwracanej z funkcji z tym, co zostało założone przez projektantów. Można również sprawdzić, czy spełnione są wymagania czasowe stawiane komunikacji między komponentami.

3. Język LML

3.1. Komponent i jego składniki

Język LML pomaga przede wszystkim w określaniu komponentów, jakie mają być poddane rejestrowaniu podczas testów integracyjnych, jak również w definiowaniu, w jaki sposób rejestrowanie ma być przeprowadzone.

Komponent w języku LML składa się z ciała i interfejsów. Ciało komponentu składa się ze zbioru plików z jednostkami kompilacji języka C, natomiast interfejsy komponentu zawierają zbiór nazw plików nagłówkowych języka C, które zawierają funkcje interfejsowe o zakresie widoczności extern. Do każdego z komponentów przypisany jest identyfikator, tak samo jak do każdego z plików ciała komponentu i plików nagłówkowych z funkcjami interfejsowymi.

W ciele komponentu znajdują się określone ścieżki dostępu dla każdej jednostki kompilacji, natomiast dla każdego pliku nagłówkowego z interfejsem komponentu należy określić jego nazwę, taką samą, jaką podano przy dyrektywach #include preprocesora języka C w jednostkach kompilacji zawierających te pliki nagłówkowe. Dzięki tak ogólnej definicji komponentu możliwe jest opisywanie szerokiej klasy komponentów w różnych sytuacjach, jakie mogą wystąpić podczas testowania integracyjnego. Jest ona niezależna zarówno od konkretnej architektury oprogramowania, jak i od decyzji projektowych. Poniższa gramatyka określa składnię języka LML dla definicji komponentu:

```
component::=component IDENTIFIER {interface_part body_part}
                                    | component IDENTIFIER {body_part interface_part}
                                    | component IDENTIFIER {body_part}
                                  interface_part::=interface {file_list}
                                    | interface {}
                                  body_part::=body {file_list}
                                  file_list::= file
                                    | file_list file
                                  file::=IDENTIFIER COLON STRING_LITERAL;
                                    | IDENTIFIER COLON @ STRING_LITERAL;
```

W definicji komponentu część określająca ciało komponentu jest obligatoryjna, natomiast część odpowiedzialna za interfejs jest opcjonalna. Lista ścieżek do jednostek kompilacji specyfikuje ciało komponentu. Format zapisu tych ścieżek jest zależny od implementacji procesora (translatora) języka LML. Jednostki kompilacji mogą być poprzedzone przedrostkiem @, który oznacza, że procesor języka LML powinien automatycznie wyszukać wyspecyfikowany plik.

3.2. Zbiór rejestrowany

Zbiór rejestrowany w języku LML oznacza zbiór wszystkich wywołań funkcji pomiędzy wywołującym a wywoływanym, jakie mają być poddane rejestrowaniu. Wywoływanym może być np. komponent, plik ciała komponentu lub konkretna funkcja zawarta w jednostkach kompilacji ciała komponentu. Każdy zbiór rejestrowany określa się jednoznacznie przez parę (wywołujący, wywoływany) oraz ich wzajemną relację. W języku LML może bardzo łatwo określić swoje intencje dotyczące tego, co powinno być rejestrowane oraz może określać w niektórych przypadkach sposób rejestrowania.

Każdy zbiór rejestrowany może być powiązany z regułami określającymi sposób rejestrowania dla wszystkich wywołań funkcji zawartych w tym zbiorze. Reguły te dotyczą sposobu instrumentacji kodu oraz specyficznych sytuacji, jakie czasem występują podczas używania konstrukcji języka C. Reguły, które mogą być powiązane z całym zbiorem rejestrowanym, nazywamy regułami ogólnym (ang. *general rules*). Są to następujące reguły:

- LOG_ALL – rejestruje wszystkie wywołania funkcji należące do zbioru przypisanego do tej reguły wraz ze wszystkimi parametrami wejściowymi i wyjściowymi, a także z wartością zwracaną,
- LOG_SMALL – rejestruje tylko same zdarzenia wywołania funkcji.

Reguły związane z wywoływaną funkcją są następujące:

- LOG_IN_PARAMS – umożliwia rejestrowanie wszystkich parametrów wejściowych funkcji, tzn. rejestrowanie następuje przed wywołaniem danej funkcji.
- LOG_OUT_PARAMS – umożliwia rejestrowanie wszystkich parametrów wyjściowych funkcji, tzn. rejestrowanie następuje zaraz po wywołaniu danej funkcji.
- LOG_RETURN_VALUE – umożliwia włączenie lub wyłączenie rejestrowania wartości zwracanej z funkcji. Używa się do tego parametrów YES/NO, które odpowiednio oznaczają, że rejestrowanie będzie włączone i wyłączone.
- LOG_AREA – umożliwia rejestrowanie żądanej liczby elementów tablicy przekazanej przez wskaźnik.
- LOG_AREA_VAR – umożliwia rejestrowanie parametru przekazanego do funkcji przez wskaźnik.

Powyższe reguły pozwalają definiować sposób rejestrowania informacji, które mogą się wydawać interesujące z punktu widzenia testów integracyjnych. Reguły języka LML wpływają na instrumentację kodu źródłowego, a każda reguła musi zostać przypisana do odpowiednich schematów instrumentacji kodu źródłowego w procesorze języka LML. Język LML można w przyszłości oczywiście rozszerzyć o nowe reguły. Pozwoli to na pomiar czasu zajętości procesora (mikrokontrolera).

3.3. Pliki języka LML

Komponenty i zbiory rejestrowane specyfikuje się w dwóch osobnych plikach. Pierwszym z nich jest plik CDF (ang. *Component Definition File*), zawierający informacje na temat deklarowanych komponentów. Stanowi on formę dokumentacji testów integracyjnych. Poprawnie jest deklarować tylko jeden taki plik dla całego oprogramowania, które jest poddawane testowaniu integracyjnemu. Drugi z nich to plik LSF (ang. *Logging Set File*), który zawiera deklaracje zbiorów rejestrowanych. Powinien on zawierać wszystkie zbiory rejestrowane, wykorzystywane do testowania danej sekwencji komunikacyjnej. Dzięki temu można ograniczyć instrumentację kodu źródłowego tylko do miejsc, w których jest to rzeczywiście potrzebne. Poniżej znajduje się przykładowe użycie języka LML w plikach CDF i LSF.

Plik CDF:

```
component PSYNC
{
   body
   {
                                    body_file : „psync.c”;
   }
   interface
   {
                                    PSYNC_API : „psync.h”;
   }
}
component MSYNC
{
   body
   {
                                    body_file : „msync.c”;
   }
}
```

Plik LSF:

```
LOG_SMALL (MSYNC.body_file -> PSYNC.PSYNC_API.foo)
```

3.4. Narzędzie do analizy kodu źródłowego

Narzędzie ITAG jest przykładem implementacji procesora języka LML. Powstało w Centrum Technicznym Delphi Poland SA w Krakowie. Jego twórcą jest Dominik Hura współpracujący z zespołem Radio Systems Software Group [3, 4]. ITAG jest to automatyczny instrumentator kodu źródłowego na potrzeby testów integracyjnych dla oprogramowania napisanego w języku C. Narzędzie to powstało w języku C#.

ITAG przeprowadza analizę kodu źródłowego komponentów opisanych w pliku CDF, do których istnieje odniesienie w zbiorach rejestrowanych zadeklarowanych w pliku LSF. Dalej wyszukiwane jest każde wywołanie funkcji należących do zbiorów rejestrowanych. Gdy już wszystkie te wywołania zostaną odnalezione, wówczas następuje generowanie kodu źródłowego, który jest następnie dodawany do komponentów. Wywołania funkcji, które zostały zarejestrowane, zamieniane są na wywołania zastępczych funkcji dodawanych do kodu źródłowego przez procesor języka LML. Dodawane funkcje zastępcze zawierają wygenerowany kod rejestrujący dla reguł związanych z konkretnym zbiorem rejestrowanym. Do kodu tego należą wywołania funkcji modułu rejestrującego, który zajmuje się przesyłaniem danych przez odpowiedni interfejs sprzętowy urządzenia.

4. Podsumowanie

Język LML jest nie tylko językiem do wspomagania testowania integracyjnego dla systemów wbudowanych, lecz także może pomóc w procesie uruchamiania oprogramowania tych systemów. W przypadku systemów czasu rzeczywistego, w których zwykle nie mamy żadnych mechanizmów wglądu w działanie oprogramowania, jest to cecha bardzo pożądana. Koncepcja zbioru zarejestrowanego może być również zastosowana do innych języków programowania strukturalnego, a także do języków zorientowanych obiektowo. Oczywiście musi ona zostać zmodyfikowana na potrzeby tych języków, tak samo jak definicja komponentu. Generacja kodu źródłowego do rejestrowania działania oprogramowania na podstawie zbiorów rejestrowanych może być ciekawą alternatywą dla istniejących systemów weryfikacji np. typu JUnit. Za pomocą rejestrowania przebiegu działania oprogramowania można zbierać informacje o wykonywaniu programu w celu późniejszej weryfikacji jego działania.

Bibliografia

1. Andrews J.H., Zhang Y.: General Test Result Checking with Log File Analysis. IEEE Trans. on Software Eng. 2003, vol. 29, no. 7, p. 634-648.
2. Andrews J.H.: Theory and Practice of Log File Analysis. Technical Report 524, Dept. Of Computer Science, University of Western Ontario, 2008.
3. Hura D.: Język zarządzania automatyczną instrumentacją kodu źródłowego podczas testowania integracyjnego oprogramowania wbudowanego, [w:] Inżynieria oprogramowania - od teorii do praktyki. Praca zbiorowa pod redakcją Zbigniewa Huzara i Zygmunta Mazura, Polskie Towarzystwo Informatyczne, Warszawa 2008.
4. Hura D.: Testy integracyjne oprogramowania wbudowanego według standardu SPICE Automotive, [w:] Od modelu do wdrożenia – kierunki badań i zastosowań inżynierii oprogramowania. Praca zbiorowa pod redakcją Włodzimierza Dąbrowskiego i Andrzeja Stasiaka, WKiŁ, Warszawa 2009.
5. Hura D., Dimmich M.: A Metod Facilitating Integration Testing of Embedded Software. Proc. of the Ninth Int. Workshop on Dynamic Analysis WODA'11, Toronto 1.07.2011, ACM, New York 2011, p. 7-11.
6. Introduction to Instrumentation and Tracing, [online], http://msdn.microsoft.com/en-us/library/x5952w0c.aspx (dnia 17.05.2012).
7. Testowanie integracyjne, [online], http://www.qatester.pl/testowanie/testowanie-w-praktyce/84-testowanie-integracyjne.html (dnia 17.05.2012).
8. Embedded Systems, [online], http://pl.sii.eu/en/our-services/engineering/embedded-systems-engineering (dnia 17.05.2012).

Abstract

The paper presents the LML language. This is a simple, declarative language for managing the automatic instrumentation of source code for recording the course of the software used Turing the integration testing of embedded software, written in the C language. There is also included an en ample of grammar of the language for each of its elements, and the way how to use it. Expression of LML language helps to reduce the time required for source code instrumentation, and also helps in describing the components of software components. These were also the LML language files, where are stored components and interfaces. The paper introduces the concept of registered collection as all the function calls between the callingand caused to be recorded. These were also prezent, the registering in the integration testing in embedded systems. The paper shows an example of implementation of the language processor LML (ITAG tool) as well as conducted a brief discussion of suitability of language LML.