2011 International Joint Conference of IEEE TrustCom-11/IEEE ICESS-11/FCST-11

Tool Support for Automated Traceability of Test/Code Artifacts in Embedded Software Systems

Christian Wiederseiner, Vahid Garousi, Michael Smith

Department of Electrical and Computer Engineering, University of Calgary
2500 University Drive NW, Calgary, AB Canada T2N 1N4
christian.wiederseiner, vgarousi, smithmr@ucalgary.ca

Abstract— **Development, testing and maintenance of software for embedded systems is a complex task. Analysis of the traceability between different software artifacts (e.g., source code, test code and requirements) is an enabling capability for better development, testing and maintenance of software systems. However, there is a general lack of tool support for automating traceability analysis for embedded systems. We demonstrate in this paper how to extend an existing unit test and test coverage framework to produce a hard-ware assisted tool framework capable of automatically deriving and visualizing traceability links between source code and test code artifacts. To demonstrate the applicability and usefulness of the framework, we report the application of the framework on realistic embedded software for vehicle gear transmission control built and deployed on the Analog Devices Blackfin ® ADSP-5XX family of DSP processors.**

I. INTRODUCTION

A vast majority of consumer products contain embedded software, for example: TVs, mobile phones, cars, airplanes, and medical systems [1]. Because of rapid changes in this field, future products will likely contain even more embedded software.

Development, testing and maintenance of embedded software systems is a complex task [1-3]. Embedded software engineering [1, 3] is an emerging and active area for both researchers and practitioners that aims at addressing various challenges in this domain, e.g., requirements engineering, reengineering and reverse engineering of existing systems, quality, maintenance, program comprehension, management of development process.

Analysis of traceability between different software artifacts (e.g., source code and test code) is a widely used practice in the embedded industry [1] and an enabling factor for better development, testing and maintenance of embedded systems.

In its beginnings, traceability in embedded software development was driven mainly by obligatory regulations such as DoD 2167a/MIL-STD-498 for US military systems [4]. Recent, quality standards now recommend traceability e.g., IEEE Standard #1219, ISO 9000ff, ISO 15504, and SEI CMM/CMMI.

Today, traceability has become a recognized attribute of software quality. For example, in a recent paper by Rummler et al. [5], the authors report about a quality certification audit of a large software development company. In that audit, poor traceability was explicitly criticized by the audit team. The authors also report that a large fraction of customers of the same company complained about missing traceability.

In a survey of eight European embedded system companies (including major firms such as Nokia and Philips), Graaf et al. [1] found that embedded software engineers see traceability among software artifacts as an essential aspect of embedded software development and project management. To keep all development products and documents consistent, the project engineers had to analyze the new features' impact and manually by utilizing automated tools. However, in the projects surveyed it was reported that the relationship between requirements and other artifacts were frequently not explicitly documented which made impact analysis difficult. Tracing artifacts to each other was difficult because the relations were too complex to specify manually (for example, between requirements and architectural components) [1]. In [6] it was noted that when *interdependencies between artifacts are documented and maintained, proposed changes of earlier (later) artifacts can be traced to necessary changes in downstream (up-steam) artifacts, supporting product development time and cost.*

The industrial survey reported in [1] also revealed that the embedded system companies need more specific, yet flexible, development and support tools (including traceability tools).

In our current on-going Collaborative Research and Development (CRD) project funded through the Canadian government, we are finding that our industrial embedded systems partners are interested in traceability tools for their projects. By searching in the literature and also the software products (either open-source or commercial), we have found a general lack of tool support for automating traceability analysis in the context of the embedded software systems.

To address the above need, we have recently designed and built a tool-set to automatically derive and visualize traceability links between source code and test code artifacts in embedded software. Our development platforms include the families of DSP processors offered by Analog Devices Inc., namely Blackfin ® , SHARC ® and TigerSHARC ® [7].

Our traceability analysis tool-set is referred to as Automated Embedded Traceability Framework (*AutoETF*) constructed as an extension of an existing unit testing and test coverage framework developed by the third author and other colleagues (reported in several earlier papers [8-10]).

An advantageous novel feature of those frameworks is that they perform test coverage analysis using processor hardware features rather than software, thus they potentially offer low overhead and are efficient for embedded systems.

In this report, we report how the *AutoETF* framework has been developed, its features and uselessness, and its application/evaluation on real embedded software.

The rest of this article is structured as follows. A review of the related work and a brief background on traceability analysis is presented in Section II. An overview of those two frameworks is presented in Section III, since some knowledge about our existing embedded unit testing [9] and code coverage measurement framework [8] is important for the presentation of the *AutoETF*. *AutoETF* itself is presented in Section IV. To demonstrate the applicability and usefulness of the *AutoETF* framework, we report in Section V its application on realistic embedded software we have designed and built for automated control of vehicle gear transmission. Finally, Section VI concludes this article and points out future works.

II. Background and Related Work

The area of traceability analysis in software engineering is quite well developed and active research area. For example, a recent 2010 survey paper [6] reported extensively on wide-spread developments and usage of traceability in conventional software engineering across 202 key publications. Analysis of traceability links across software artifacts can be performed in various stages of the software development lifecycle (SDLC).

An example hypothetical traceability graph showing traceability links among artifacts in five typical phases of the SDLC is depicted in Fig. 1. In this graph, an edge (arrow) denotes a traceability link, meaning either usage of an earlier artifact in developing a later artifact, or an artifact depending on another (e.g., a test case testing a code artifact). The areas shaded by gray are the focus and contributions of the current work in the embedded system's context.

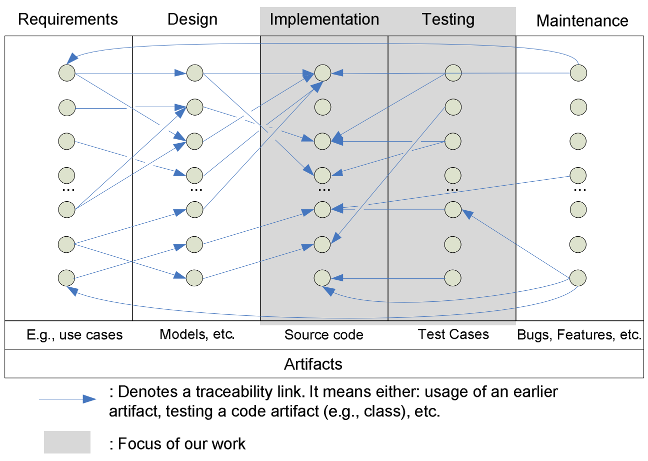


Fig. 1-An example hypothetical traceability graph showing traceability links among artifacts.

Although tools for automated traceability in the software engineering literature [6] exist, there is a lack of tool support focusing in the context of embedded software. Many companies and embedded software developers feel the need for automated tool support traceability [2].

To the best of the authors' knowledge, works in [11, 12] are the key traceability methodologies for embedded software and the toolset reported in [13] is the only tool environment to support traceability in this domain.

Von Knethen [11] has proposed a set of models and a working environment [13] in the domain of embedded systems. The approaches in [11, 13] support change-impact analysis (as an application of traceability analysis) and propagation of changes based on previously-recorded traceability links. However, the tool-set does not support automated extraction of traceability links between application and test code as proposed in our work.

III. An Overview of Our Existing Embedded Unit Test and Coverage Measurement Framework

The third author of this paper and other colleagues have developed and reported previously [8-10] an embedded unit framework (*E-Unit*) and a hardware-assisted framework for efficient code coverage analysis in embedded environments. We provide an overview of those two frameworks in Fig. 2 since our traceability framework is built on top of them. For details, the interested reader is referred to [8-10].

The diagram in Fig. 2 shows where each component is deployed (i.e., on the embedded device under test, or the desktop computer). The top figure depicts the actual connection to the embedded device (Blackfin ® DSP board in this case) of the development tool installed and running on a PC. To clarify the contributions of the current work versus [8-10], note the three shades present for the modules in the legend of the bottom figure in Fig. 2.

In our current platform, the desktop computer runs the development tool, Analog Devices VisualDSP++®, which is an Integrated Development and Debugging Environment (IDDE) for embedded software. We extended VisualDSP++® to add support for embedded unit (*E-Unit*) testing features [8] and code coverage [9].

E-Unit capabilities include creation, build, executions, and results reporting of the unit test suite running on the hardware under test. The coverage engine [9] calculates two types of test coverage metrics: function (method) and statement. The coverage framework was developed using a hardware feature in processors called the *branch vector buffer*. Use of the hardware performance monitoring unit (containing the above buffer) has been demonstrated to reduce code coverage profiling overheads by moving much of the logging process into hardware [8, 14]. While a basic version of this can be found on some PC-oriented processors such as the Itanium 2 by Intel [14], this feature has much greater flexibility and power with embedded systems where it forms a fundamental part of the embedded processor's debugging sub-system [8].

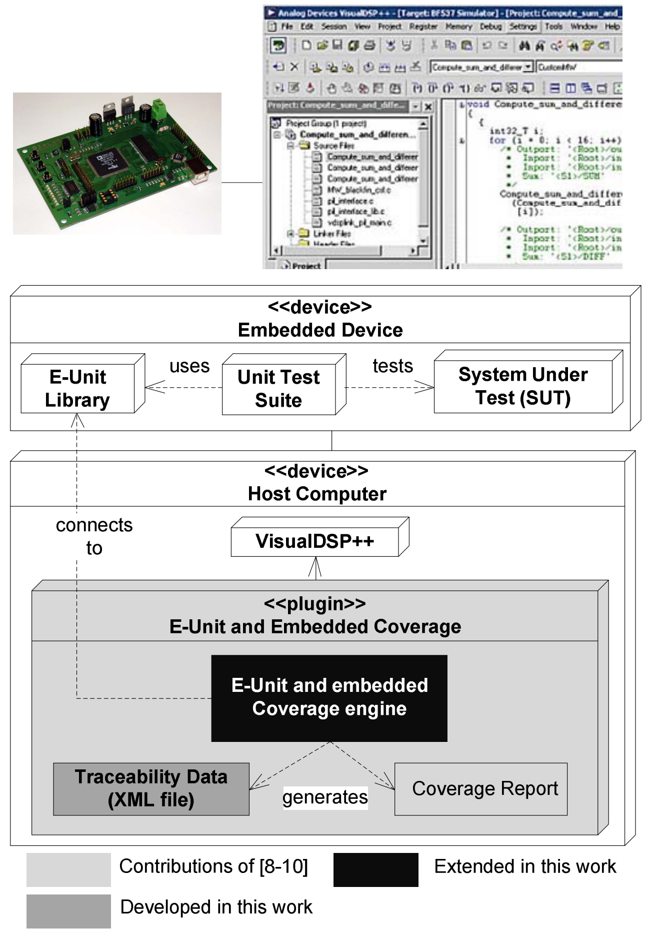


Fig. 2-(Top): Actual connection of the IDE in the PC to the embedded device (Blackfin ® DSP board in this case). (Bottom): A UML deployment diagram showing the coverage framework [8], the *E-Unit* testing framework [9], and the extensions made in this work.

The desktop computer lets the engineer build and deploy the Software Under Test (SUT) and its automated test suite using the IDDE. The E-Unit library is automatically deployed on the embedded device by VisualDSP++®.

After executing the test suite on the embedded device, the test results of the test suite are brought back from embedded device to the desktop computer using the automation API of VisualDSP++® and the results displayed in the IDDE [8]. The coverage plug-in embedded inside the IDDE then generates the test coverage reports. Of particular interest is the fact that the test coverage analysis is performed with low overhead (near real-time) using hardware features present in the Analog Device's families of DSP processors rather than the customary software analysis.

With our additions in the current work, the coverage plug-in produces also, in addition, the traceability data from each execution of test suites. In other words, the current work adds to the existing platform [8-10] an automatic traceability between test and code artifacts.

IV. Embedded Traceability Framework

Our Automated Embedded Traceability Framework (*AutoETF*) has two modules: (1) automated traceability extraction, and (2) traceability visualization tool.

A. Automated Traceability Extraction

Automated traceability extraction is the process of extracting traceability links between source and test code artifacts. Recall from Fig. 1 that traceability links can be among artifacts in different phases of the SDLC, e.g., requirements, design, development, testing and maintenance. In our first attempt in this work, we have currently developed automated extraction of traceability links between source (production) code and test code artifacts.

Fig. 3 depicts a detailed view of part of Fig. 2 showing the software tools we have developed and deployed on a desktop computer to perform automated traceability extraction. We have extended the *E-Unit* test and embedded coverage engine to extract traceability links while measuring code coverage of test suites.

Automated extraction of traceability links works as follows. During the post-build of the test suite code in the IDDE, a VisualDSP++® utility called *ElfDump* generates the symbol table of the code-base (shown as *symtab* in Fig. 3) and source code line data (*debug_line*, details in [15]) into two text files (symtab.txt and linetab.txt). After test suite executions, the *E-Unit* test and embedded coverage engine reads, using the Automation API from IDDE, addresses of the program's control flow stored in the memory of embedded device. Those addresses are mapped to functions and source lines data, using respectively *symtab.txt* and *linetab.txt* data to create test/code traceability links.

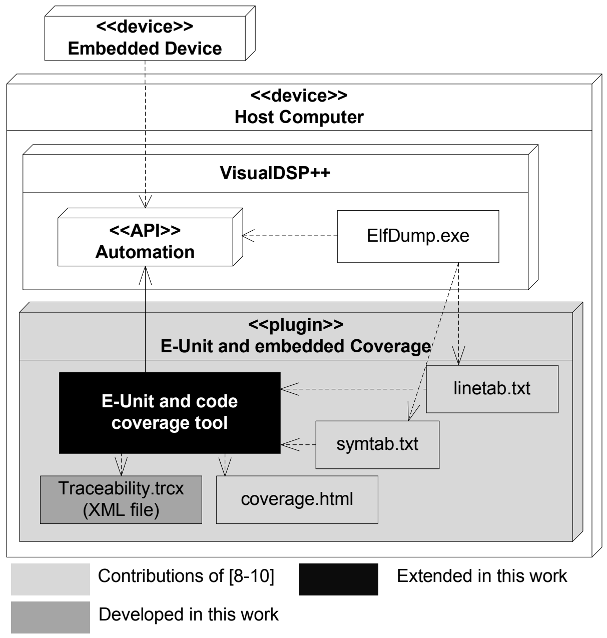


Fig. 3- Detailed view of part of Fig. 2 showing the software tools deployed on desktop computer that perform automated traceability extraction.

Building traceability links between artifacts is realized by identifying the position (start, inside or end) of methods to determine when each test artifact calls a source code artifact. The extended embedded coverage engine generates an XML file containing traceability links between source code and test code. Since our next step is to visualize the traceability links using a graph visualization framework (Gephi [16]), we have designed the embedded coverage engine in a way that the XML file is created in the Graph Exchange XML Format (GEXF) [17].

For traceability links between artifacts, different granularities for either source code or test code must be supported. To formally present this concept, a UML meta-model showing traceability links between different granularities of source (production) code and test artifacts is shown in Fig. 4. A source (production) code artifact can be a namespace, class, method, line or a target memory address. On the other hand, a test artifact can either be a test suite, test method, test line of code, or a memory address storing the code.

This hierarchical scheme allows our framework to support 20 (5x4) granularities of traceability links between the two artifact types. An example of the traceability links extracted by our tool-set is provided in the next section. A larger example and application of the *AutoETF* framework on a real large-scale embedded software is discussed in Section V.

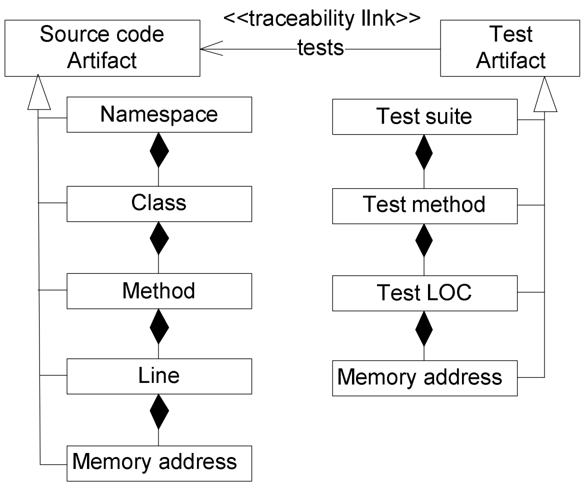


Fig. 4- A UML meta-model showing traceability links between different granularities of source code and test artifacts.

B. *Traceability Visualization Tool*

We have developed a traceability visualization tool by adapting the popular Gephi graph visualization framework [16]. The entire source code of our visualization tool is available online in the Google Code repository [18] and can be used by researchers and practitioners.

A workflow diagram showing how the traceability visualization tool works and how it is used is shown in Fig. 5.

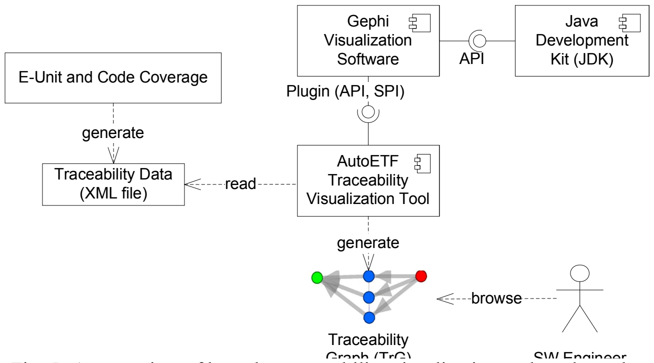

Gephi Visualization Software
API
Java Development Kit (JDK)
E-Unit and Code Coverage
Plugin (API, SPI)
generate
Traceability Data (XML file)
read
AutoETF Traceability Visualization Tool
generate
browse
Traceability Graph (TrG)
SW Engineer

Fig. 5- An overview of how the traceability visualization tool works and how it is used.

With the XML file containing traceability links among the source and test artifacts generated, our traceability visualization tool is used to visualize the traceability data as a traceability graph (TRG).

```
int Addition::addition(int a, int b){
    return a+b;
}

int Subtraction::subtraction(int a, int b){
    return a-b;
}
```

Fig. 6- Source (production) code of two simple classes

```
TEST(additionPositiveValues)
{
  CODE_COVERAGE_START_LOGGING(); //macro
  Addition a;
  int output = a.addition(5,6);
  CODE_COVERAGE_STOP_LOGGING(); //macro
  CHECK_EQUAL(11, output);
}
TEST(additionNegativeValues)
{
  CODE_COVERAGE_START_LOGGING();
  Addition a;
  int output = a.addition(-5,-6);
  CODE_COVERAGE_STOP_LOGGING();
  CHECK_EQUAL(-11, output);
}
TEST(subtractionPositiveValues)
{
  CODE_COVERAGE_START_LOGGING();
  Subtraction s;
  int output = s.subtraction(5,6);
  CODE_COVERAGE_STOP_LOGGING();
  CHECK_EQUAL(-1, output);
}
TEST(subtractionNegativeValues)
{
  CODE_COVERAGE_START_LOGGING();
  Subtraction s;
  int output = s.subtraction(-5,-6);
  CODE_COVERAGE_STOP_LOGGING();
  CHECK_EQUAL(1, output);
}
```

Fig. 7-Four example test cases (methods) for code covered classes `Addition` and `Subtraction`

As an example of a TRG, let us present simple embedded software with two classes: `Addition` and `Subtraction`. Code listing for it is shown in Fig. 6. Four example test cases (methods) for these two classes are shown in Fig. 7. Two CODE_COVERAGE macros in Fig. 7 start and stop test (code) coverage and traceability.

As mentioned above, a TRG can be generated for different granularities of source code and test artifacts. Two example TRGs rendered in our visualization tool for the above example system are shown in Fig. 8. In both TRGs, the test artifact granularity is test method. In the top and bottom TRG, the source code artifact granularities are methods and lines of code, respectively.

Note that our traceability visualization tool has automatically generated these example TRGs and the user can perform standard graph browsing tasks such as zooming, panning, moving nodes around, etc. to understand how different artifacts are related and trace to each other.

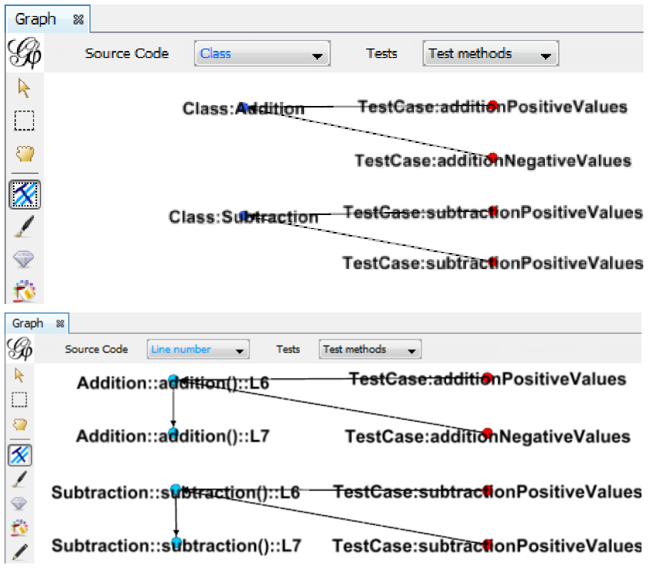

Graph
Source Code
Class
Tests
Test methods
Class:Addition
TestCase:additionPositiveValues
TestCase:additionNegativeValues
Class:Subtraction
TestCase:subtractionPositiveValues
TestCase:subtractionPositiveValues
Graph
Source Code
Line number
Tests
Test methods
Addition::addition()::L6
TestCase:additionPositiveValues
Addition::addition()::L7
TestCase:additionNegativeValues
Subtraction::subtraction()::L6
TestCase:subtractionPositiveValues
Subtraction::subtraction()::L7
TestCase:subtractionPositiveValues

Fig. 8-Two example Traceability Graphs (TRG) corresponding to the `Addition` and `Subtraction` example (Fig. 6 and Fig. 7).

C. *Making Use of Traces (i.e., Traceability Links)*

As Winkler and von Pilgrim discuss in their survey paper [6], recording, extracting, storing and visualizing software traces only makes sense if the traces can be used later. In fact, a project's traceability goals should generally drive the activity of extracting traces.

Winkler and von Pilgrim aggregate [6] from the existing traceability literature that traceability links have been used for many purposes in software engineering including the following (refer to [6] for references to articles on each type of usage): estimating change impact, showing system's adequateness, validating artifacts, supporting audits, improving changeability, monitoring progress, assessing the development process, understanding the system, documenting reengineering, and finding reusable elements.

In our own industrial collaborations, we have frequently observed that software engineers and testers (especially in the embedded context) have considerable challenges in the following scenarios:

1. Test coverage (adequacy) improvement (i.e., white-box testing)
2. Test suite maintenance as the software under test (SUT) evolves
3. Fault localization
4. Test redundancy detection

Our proposed *AutoETF* framework aims at addressing all the above challenges as is discussed next. For usage scenario #1 above (test coverage improvement), testers need to find the uncovered parts of the SUT and attempt to cover them by creating new test cases. For this purpose, testers only require to find the SUT nodes with no incoming edges when using TRG. Using the conventional code-coverage tools to show the coverage information, multiple files should need to be inspected manually for this purpose so that extensive time from human testers is required to find uncovered parts of the SUT.

As an example of the usage scenario #2 above, consider the following scenario. As a result of SUT evolution, source code modifications can make a number of test methods invalid for regression testing and thus testers need to modify them appropriately. TRG can help testers to find all the tests related to the modified part of the system. For this purpose, the tester needs to find the modified SUT node in the graph and consider all those tests which are covering these items.

In the usage scenario #3 above by referring to TRGs, testers can find which part of the system is covered more frequently by failing tests. The number of incoming edges to a SUT item and the number of the calls shown on each of those edges can be used to find more suspicious fault prone parts of the SUT [19]. Again, using the conventional code-coverage tools to show the progress-bar like coverage information, multiple files must be inspected manually at extensive effort.

To detect redundant test cases (usage scenario #4 above), testers need to find the tests that cover part of the system which are also covered by other test cases in the test suite. For this purpose, TRG can be useful to find which parts of the system are covered by each test and compare the covered parts by various test cases with each other.

Last but not least, from a higher-level perspective, we can qualitatively estimate the cost of test maintenance (entailed by a change in the SUT) by visually analyzing the volume of edges from the test suite domain to the SUT domain.

V. Evaluation on an Embedded Controller Software for Vehicle Gear Transmission (EmbeddedGear)

To demonstrate the applicability and usefulness of our traceability framework, we applied the tool on a realistic embedded software that we have developed for vehicle gear transmission control (referred to as *EmbeddedGear* in the rest of paper). The software was built and deployed on Blackfin ® ADSP-BF533, BF561 and BF548 DSP processors [7] made by Analog Devices Inc.

We should mention that we would have preferred to find a realistic embedded software already developed by others (e.g., commercial or open-source) compatible with our Blackfin ® platform. However, after spending considerable amount of time, we were only able to find several small-scale software developed by Analog Devices Inc., e.g., a small C program (function) to do the Fourier transforms (a type of signal processing), for learning purposes. Those examples are much smaller scale compared to the EmbeddedGear system that we have developed.

We discuss next an overview of the EmbeddedGear software, its automated test suite, our initial attempt to conduct traceability analysis of this system, and as a result of our initial evaluations, some discussions on the usefulness of *AutoETF* for development, testing and maintenance of this system.

A. An Overview of the EmbeddedGear Software

The Matlab Simulink platform provides as a demo (tutorial) a fully-implemented and executable model for automotive power transmission control [20]. We reverse engineered this model and used it as the requirements and design document to develop the EmbeddedGear system.

A typical automotive power train system is shown in Fig. 9 [20]. It is composed of the following four blocks: engine, transmission, transmission control unit and vehicle dynamics.

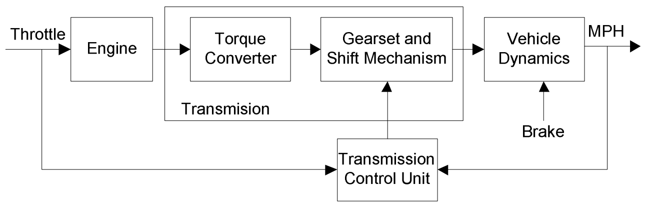


Fig. 9. A typical automotive power train (adapted from [20]).

Transmission itself consists of two blocks: torque converter and gear-set shift mechanism. The transmission block converts the mechanical motion into a motion that drives the wheels. Transmission control unit automatically changes the gears. Vehicle dynamics takes into account external factors that slowdown the car such as brake pedal, wheels, road and air friction.

The UML use case diagram of EmbeddedGear is shown in Fig. 10 (reverse engineered from the Matlab Simulink model [20]). Driver has the ability to manipulate the throttle and brake only. The automatic transmission control unit will then set the appropriate gear, will update the vehicle speed and set the engine RPM (revolutions per minute).

The class diagram of the EmbeddedGear software is shown in Fig. 11. There are eight thread classes which inherit from the `Thread` class in the VisualDSP++ Kernel (VDK) library. Thread classes are responsible for parallel execution of different components (objects) of EmbeddedGear system, and input/output synchronization of their signals.

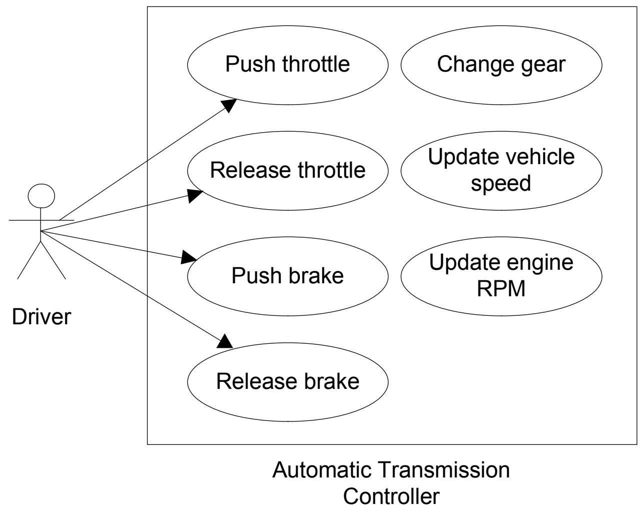


Fig. 10. UML use case diagram of EmbeddedGear.

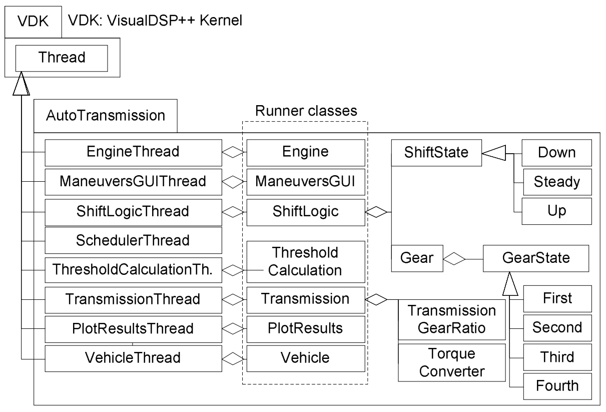


Fig. 11. Class diagram of the EmbeddedGear software.

Each thread class (except `SchedulerThread`) has another (runner) class inside itself which is executed by the thread class. Class `SchedulerThread` itself acts a runner class and an object of this class performs the scheduling of all the other threads by setting execution times among different objects.

The main logic of the transmission control is carried by classes `Gear`, `GearState`, `TransmissionGearRatio` and `TorqueConverter`. The `GearState` class has four child classes denoting gears 1 to 4 (we have only modelled a 4-gear transmission system in this software, but more gears can be added).

The `ShiftState` class has three child classes (designed based on the state design pattern [21]) denoting the shifting states down and up and also the steady state for no shifting.

State charts for two stateful classes of the EmbeddedGear `ShiftState` and `GearState` are shown in Fig. 12.

These diagrams have been reverse engineered from the Matlab Simulink executable models in [20].

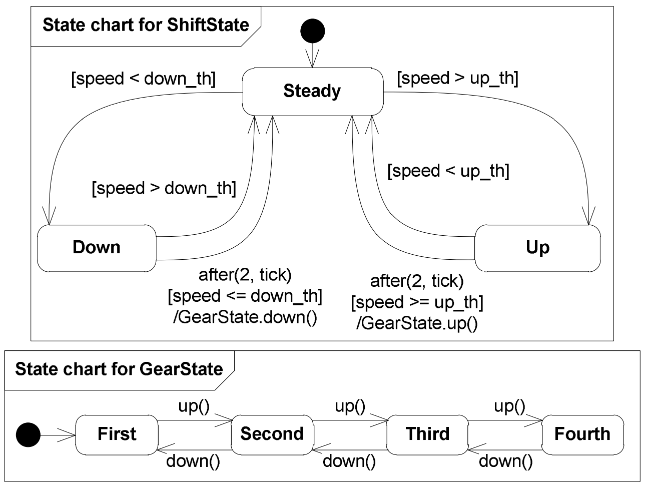


Fig. 12. State charts for classes `ShiftState` and `GearState`.

Some brief statistics about the source-code base of the tool are provided in Table 1. The C/C++ code counter tool CCCC [22] was used to generate these statistics. Number of classes are 28 (the same in the class diagram). EmbeddedGear source code size is currently 1,509 C++ LOC. The entire source code of EmbeddedGear is available online in the Google Code repository [23].

Table 1- Statistics of the EmbeddedGear's source-code base

Metric	**Overall**	**Per Class**
Number of Classes	28	
Lines of Code (excluding comments)	1,509	53.8
Line of Comments	712	25.4
McCabe's Cyclomatic Number	286	10.2

B. *An Overview of the Automated Test Suite of the EmbeddedGear*

To verify and validate the behavior of EmbeddedGear, an automated test suite (in unit and integration test levels) was developed using the *E-Unit* testing framework [9].

The test code (developed in C++) was 2,436 LOC in total, consisting of 413 test methods (cases). Recall from software testing foundations [24] that test case generation can be done based on three approaches: (1) black-box, (2) white-box (based on code coverage), (3) and gray box (combination of approaches 1 and 2). As it is usually conducted in automated testing activities [24], we started with a black-box approach to test case generation.

From among different black-box test techniques, we selected the category partitioning technique, a widely practiced test approach [24]. In category partitioning, the input space of each method in SUT class was partitioned such that the expected behavior of the method for all values in a partition was deemed the same (based on requirements) [24].

The other black-box test approach we used was contract-based testing [25]. Recall from Section V.A. that to develop EmbeddedGear, we used an executable Matlab Simulink model [20] as the system requirements for EmbeddedGear. Based on the Matlab Simulink model, we derived contract-based test cases in two levels: unit testing and integration testing. In the unit level, test code was written to exercise each function and as the expected output, we used the values generated by the Matlab Simulink gear system model [20]. In the integration testing level, test code was written to exercise several classes (as a subsystem) and the subsystem's behavior was compared with the output of the Matlab Simulink model [20].

Breakdown of the number automated tests for each class of the EmbeddedGear generated by each of the above three black-box test approaches is provided in Table 2. To manage the efforts on creating test cases for each classes, we used the guidelines of value-based software testing [26], i.e., classes under test were prioritized so that high priority classes would have more test cases.

Table 2- Information about the Automated Unit Test Suite of EmbeddedGear

Class under test	Type of test generation activity			
	Category partitioning	Contract-based unit testing	Contract-based integration testing	Sum
`Engine`	47	102	6	155
`Vehicle`	0	53	0	53
`ShiftLogic`	1	0	6	7
`Gear`	1	0	2	3
`ThreasholdCalculation`	48	8	0	56
`TorqueConverter`	0	69	9	78
`Transmission`	0	0	20	20
`TranmissionGearRatio`	0	37	4	41
Total number of test cases	97	269	47	413

In fact, the AutoETF has the potential to help software testers in white-box testing by enabling them to derive test cases based on test (code) coverage. In other words, using the AutoETF, if a tester analyzes a visual TRG, s/he can detect code artifacts not covered by any test case, and thus can decide to develop new test cases to cover those parts of the code. Due to the scope of our project, however, we did not conduct white-box testing for the EmbeddedGear system.

C. *Traceability Analysis of EmbeddedGear*

We applied our toolset to the EmbeddedGear by running its test suite on a Blackfin ® model BF548 DSP processor [7] and used our tool to visualize several types of traceability graphs (TRGs).

Due to space constraints, it is not possible to report all combinations of different granularities of a TRG in this paper. Thus we only report four selected types of TRGs here

(Fig. 13 and Fig. 14). To provide more details, we have prepared a screen-cast video demo of traceability analysis of EmbeddedGear using our visualization tool which can be viewed online at [27].

To provide a few examples, Fig. 13 and Fig. 14 depict three selected TRGs for the classes `Engine` and `Transmission` as two important classes in this system. In Fig. 13, the granularity of the SUT artifacts (in left side) are methods and lines (of class `Engine`) in the top and bottom TRGs, respectively. In both TRGs, test artifacts are unit test methods.

In the TRG shown in Fig. 14, the granularity of the SUT artifacts (in left side) is methods (of class `Transmission`). Test artifacts are unit test methods. Usefulness of the AutoETF framework based on these three TRGs is discussed next. The width of the edges in a TRG denotes the frequency of calls from the involved test artifact to the code artifact.

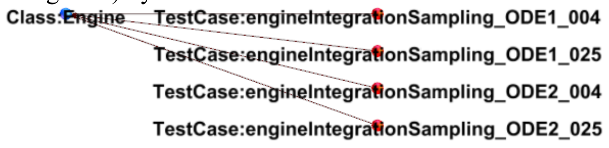
D. *Usefulness of AutoETF*

After developing the AutoETF framework, our goal was to evaluate its usefulness based on the usage scenarios of traceability links from the literature [6] and also based on our own experience in working with industrial partners.

During development, testing and maintenance of EmbeddedGear, we extensively used different views of TRGs to facilitate the tasks on hand. More specific usage scenarios were test coverage improvement and test suite maintenance as the software under test (SUT) evolved. For example, the TRG shown in Fig. 14 shows that only six of the methods of class `Transmission` are covered by the existing test suite. Thanks to the information provided by these TRG, we were able to effectively develop new test code to cover the other uncovered methods.

Although we have not yet conducted extensive case studies and/or experiments to quantitatively measure the advantages of the AutoETF, our initial observations have revealead that the AutoETF has the potential to greatly benefit embedded software engineers by reducing their development, testing, and maintenance efforts.

Traditional code coverage information (in text and number format, e.g., 80%) can be of help in this challenge. However we have found in the current work and also a previous study [19] that a visual representation of coverage is better for these purposes since it can provide a high-level bird-eye view for the test engineer on both the SUT code and test artifacts without overwhelming the test engineer with details. To support this claim, a controlled experiment (reported in [19]) was conducted previously in our research group and it empirically found that visual representation of coverage increased the efficiency of a selected team of software testers (testing time) by about 304%.


Class:Engine
TestCase:engineIntegrationSampling_ODE1_004
TestCase:engineIntegrationSampling_ODE1_025
TestCase:engineIntegrationSampling_ODE2_004
TestCase:engineIntegrationSampling_ODE2_025

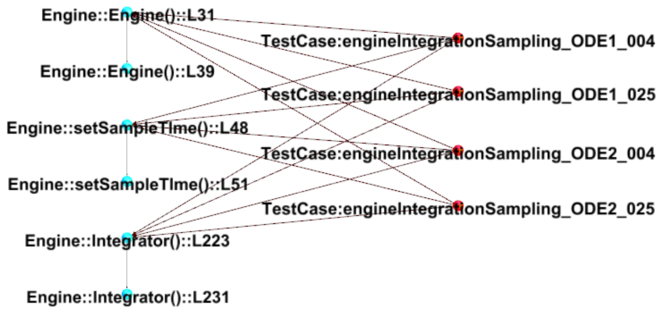

Engine::Engine()::L31
Engine::Engine()::L39
Engine::setSampleTime()::L48
Engine::setSampleTime()::L51
Engine::Integrator()::L223
Engine::Integrator()::L231
TestCase:engineIntegrationSampling_ODE1_004
TestCase:engineIntegrationSampling_ODE1_025
TestCase:engineIntegrationSampling_ODE2_004
TestCase:engineIntegrationSampling_ODE2_025

Fig. 13. Two TRGs for the class `Engine`.

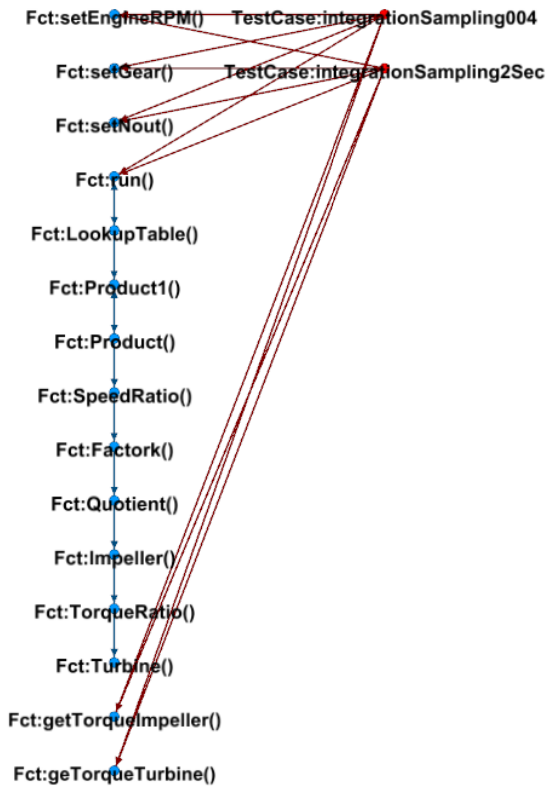


Fig. 14. A TRG for class `Transmission`.

VI. CONCLUSIONS AND FUTURE WORKS

We presented in this paper a tool framework to automatically derive and visualize traceability links between source code and test code artifacts in embedded software. To demonstrate the applicability and usefulness of the framework, we reported the application of the tool on an embedded software called EmbeddedGear.

To verify and validate the functionality of our *AutoETF* traceability framework, although we have not yet conducted rigorous testing (e.g., by developing automated test suites for the framework itself), we have conducted a large number of manual inspections and verification activities using several example SUTs (e.g., EmbeddedGear). As a future work, we plan to develop automated test suites for the *AutoETF* framework itself. This will especially be useful for regression testing of the framework itself as it evolves.

The traceability framework has been helpful for our project so far in development, testing and maintenance of EmbeddedGear, e.g. in test coverage improvement and test suite maintenance. As a future work, we plan to conduct more usage scenarios of the proposed traceability framework based on the needs of our industrial partners. We also plan to conduct more measurable case studies and/or controlled experiments to objectively study the usefulness of our framework to embedded software engineers in their development, testing, maintenance tasks.

ACKNOWLEDGMENT

This project was financially supported by the Discovery Grant no. 341511-07 and also the Collaborative Research and Development (CRD) grant no. 365295-08, provided by the Natural Sciences and Engineering Research Council of Canada (NSERC) and also by Analog Devices Inc.

REFERENCES

[1] B. Graaf, M. Lormans, and H. Toetenel, "Embedded software engineering: the state of the practice," *IEEE Software,* vol. 20, pp. 61-69, 2003.

[2] L. Murray, A. Griffiths, P. Lindsay, and P. Strooper, "Requirements Traceability for Embedded Software - an Industry Experience Report," in *Int. Conf. on Software Engineering and Applications*, 2002, pp. 374-068.

[3] E. A. Lee, "What's ahead for embedded software?," *IEEE Computer,* vol. 33, pp. 18 - 26, 2000.

[4] M. Jarke, "Requirements tracing," *Commun. ACM,* vol. 41, pp. 32-36, 1998.

[5] A. Rummler, B. Grammel, and C. Pohl, "Improving traceability inmodel-driven development of business applications," in *ECMDA Traceability Workshop*, 2007, pp. 7-15.

[6] S. Winkler and J. von Pilgrim, "A survey of traceability in requirements engineering and model-driven development," *Software and Systems Modeling,* vol. 9, pp. 529-565, 2010.

[7] A. D. Inc.;, "Processors and DSP," www.analog.com/en/processors-dsp/products, Last accessed: June 2011.

[8] A. Tran, M. Smith, and J. Miller, "A Hardware-Assisted Tool for Fast, Full Code Coverage Analysis," in *International Symposium on Software Reliability Engineering*, 2008, pp. 321-322.

[9] M. Smith, A. Kwan, A. Martin, and J. Miller, "E-TDD (Embedded Test Driven Development): A Tool for Hardware-Software Codesign," in *Proc. of International Conf. on eXtreme Programming and Agile Processes*, 2005, pp. 145-153.

[10] M. Smith, J. Miller, L. Huang, and A. Tran, "A More Agile Approach to Embedded System Development," *IEEE Software,* vol. 26, pp. 50-57, 2009.

[11] A. von Knethen, "Change-Oriented Requirements Engineering. Support for Evolution of Embedded Systems," *Ph.D. thesis, Universität Kaiserslautern, Germany,* 2001.

[12] A. Albinet, J. L. Boulanger, H. Dubois, M. A. Peraldi-Frati, Y. Sorel, and Q. D. Van, "Model-based methodology for requirements traceability in embedded systems," in *ECMDATraceability Workshop (ECMDA-TW)*, 2007, pp. 27–36.

[13] A. von Knethen and M. Grund, "QuaTrace: a tool environment for (semi-) automatic impact analysis based on traces," in *Proc. of the IEEE International Conference on Software Maintenance (ICSM)*, 2003, pp. 246–255.

[14] Intel Corporation, "Intel Itanium 2 Processor Reference Manual for Software Development and Optimization," May 2004.

[15] IBM, "DWARF/ELF Extensions Library Reference, Run-time Library Extensions, Debug section," http://publib.boulder.ibm.com/infocenter/zos/v1r11/index.jsp?topic=/com.ibm.zos.r11.cbcdd01/debugsection_srcfiles.htm, Last accessed: July 2011.

[16] M. Bastian, S. Heymann, and M. Jacomy, "Gephi: an open source software for exploring and manipulating networks," in *International AAAI Conference on Weblogs and Social Media*, 2009.

[17] GEXF Working Group, "GEXF File Format Specifications," www.gexf.net/format/schema.html, Last accessed: July 2011.

[18] C. Wiederseiner, V. Garousi, and M. Smith, "Automated Embedded Traceability Framework (AutoETF) in Google Code repository," *code.google.com/p/autoetf,* Last accessed: July 2011.

[19] V. Garousi and N. Koochakzadeh, "An Empirical Evaluation to Study Benefits of Visual versus Textual Test Coverage Information," in *Proceedings of the International Conference on Testing: Academic and Industrial Conference - Practice and Research Techniques (TAIC PART)*, 2010, pp. 189-193.

[20] The MathWorks Inc., "Simulink - Modeling an Automatic Transmission Controller," http://www.mathworks.com/products/simulink/demos.html?file=/products/demos/shipping/simulink/sldemo_autotrans.html, Last accessed: July 2011.

[21] A. Shalloway and J. R. Trott, *Design Patterns Explained: A New Perspective on Object-Oriented Design*, 2nd ed.: Addison-Wesley, 2004.

[22] T. Littlefair, "CCCC - C and C++ Code Counter," http://cccc.sourceforge.net, Last accessed: July 2011.

[23] C. Wiederseiner, V. Garousi, and M. Smith, "EmbeddedGear Software in Google Code repository," *code.google.com/p/embeddedgear,* Last accessed: July 2011.

[24] A. P. Mathur, *Foundations of Software Testing*: Addison-Wesley Professional, 2008.

[25] P. Madsen, "Testing by contract - combining unit testing and design by contract," in *Proceedings of Nordic Workshop on Programming and Software Development Tools and Techniques*, 2002.

[26] R. Ramler, S. Biffl, and P. Grünbacher, "Value-Based Management of Software Testing," in *Value-Based Software Engineering*, S. Biffl, A. Aurum, B. Boehm, H. Erdogmus, and P. Grünbacher, Eds.: Springer, 2006.

[27] C. Wiederseiner, V. Garousi, and M. Smith, "Screen-cast video demo of Traceability Analysis using AutoETF," http://vimeo.com/channels/239297, Last accessed: July 2011.