Platform Support for Developing Testing and Analysis Plug-ins

Shauvik Roy Choudhary, Jeremy Duvall, Wei Jin, Dan Zhao, Alessandro Orso
College of Computing
Georgia Institute of Technology
{shauvik|jduv|weijin|dzhao|orso}@gatech.edu

ABSTRACT

Plug-ins have become an important part of today's integrated development environments (IDEs). They are useful for extending the functionality of these environments and customizing them for different types of projects. In this paper, we discuss some features that should be provided by IDEs to support the development of a specific kind of plug-ins—plug-ins that support program analysis and software testing techniques. To guide the discussion, we leverage our experience in building a plug-in for two different platforms and generalize from that experience.

Categories and Subject Descriptors: D.2.6 [**Software Engineering**]: Programming Environments—*Integrated environments*

General Terms: Design, Standardization

Keywords: Integrated development environments, plug-ins, software testing, program analysis.

1. INTRODUCTION

The feature set offered in modern Integrated Development Environments (IDEs) has progressed significantly from the era of simple (albeit smart) text editors. As the complexity of software systems have grown, so have demands on the tools needed to build them. As a consequence, IDEs have evolved into complex environments that are designed to aid developers' productivity by providing support for a large number of software development tasks, such as coding, access to APIs, testing, and maintenance, just to cite a few. Moreover, instead of multi-tool environments, where compilers, linkers, debuggers, and testing tools are all segregated entities, many of these technologies are seamlessly integrated in modern IDE designs. Another important feature of modern IDEs is that they provide frameworks that allow for extending an IDE's core functionality and customize it to more specialized needs through the addition of plug-ins.[1]

Ideally, an IDE would provide a set of APIs that suite the needs of every flavor of external plug-in imaginable. Unfortunately, this is not possible, given the wide variety of needs of different plug-ins and the risk of overwhelming the users with an API that is too broad. It is therefore important to choose what functionality should be exported by a plug-in framework for an IDE, so as to maximize the cost benefits for plug-in developers. In this paper, we provide a basis for discussing what kind of features an IDE should provide to support a specific kind of plug-ins, that is, program analysis and software testing plug-ins. We choose to target this specific area both because of our expertise and, most importantly, because many plug-ins rely on functionality that is directly or indirectly related to some form of program analysis, whether static or dynamic. (For example, refactoring plug-ins rely heavily on program analysis to compute information such as program dependences.) We therefore believe that the topic is of interest for a broad audience among plug-ins developers.

We stress that our goal is not to be comprehensive and cover all possible aspects of this topic. We rather aim to raise some issues that can be expanded and discussed in more detail at the workshop. As a starting point for our discussion, we consider a case study—the development of a testing plug-in for two different platforms and IDEs: Eclipse [6] and Microsoft Visual Studio [4]. The case study is based on the direct experience of three of the authors of this paper, who are the main developers of the plug-in considered. The plug-in was initially developed for the Eclipse platform and was then ported to the Visual Studio platform.

The rest of the paper is organized as follows. In the next section, we present the specifics of the plug-in we use as a case study and provide an overview of its functionality. Section 3 discusses our experience in implementing our plug-in on the two IDEs considered. Next, in Section 4, we discuss some of the features that an IDE should provide for the development of analysis and testing plug-ins. Finally, Section 5 provides some concluding remarks.

2. CASE STUDY: BERT

As we mentioned in the Introduction, the case study for this work consists in a plug-in that was implemented first for the Eclipse IDE and then for Microsoft Visual Studio. The plug-in implements a technique developed by some of the authors and called BERT, which stands for BEhavioral Regression Testing [2, 3]. BERT is a recent approach designed to complement existing regression testing strategies that (a) identifies the behavioral differences between two versions of a program and (b) leverages those differences in an attempt to pinpoint potential regression faults—unforeseen side effects of a change.

BERT is designed to operate within an IDE. As soon as a developer generates a new syntactically-correct version of a program (e.g., by saving a class after modifying it), BERT is invoked and provided with the old and new versions of the program (V0 and

[1]Plug-ins can also be called add-ins (e.g., in the case of Visual Studio). To avoid confusion, we use only the term plug-in.

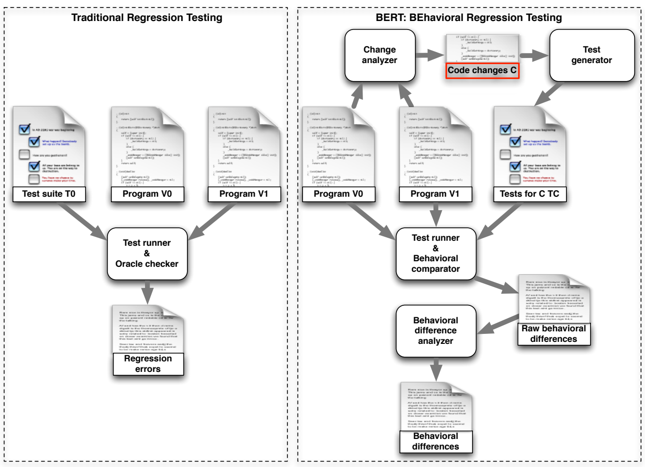
Traditional Regression Testing
Test suite T0
Program V0
Program V1
Test runner & Oracle checker
Regression errors
BERT: BEhavioral Regression Testing
Change analyzer
Code changes C
Test generator
Program V0
Program V1
Tests for C TC
Test runner & Behavioral comparator
Raw behavioral differences
Behavioral difference analyzer
Behavioral differences

Figure 1: High level view of the BERT approach.

V1, respectively). As shown in Figure 1, that compares BERT to a traditional regression-testing approach, BERT consists of four main phases, implemented by four corresponding modules: change analyzer, test generator, test runner and behavioral comparator, and behavioral difference analyzer.

The *change analyzer* compares the two versions V0 and V1 of the program and identifies which parts have changed. In the case of Java or C# code, the changes consist of the set of classes that have been modified when going from V0 to V1. The *test generator* automatically generates test cases for the changed code (e.g., using random test generation or a more sophisticated approach such as symbolic execution). The *test runner and behavioral comparator* executes the test cases produced by the test generator on V0 and V1, compares the behavior of the tests on the two versions, and records differences in such behavior. Finally, the *behavioral difference analyzer* analyzes the differences computed by the behavioral comparator to refine and rank them.

The final result of these four phases is a set of behavioral differences that are presented to the developer and ranked so as to show first changes that are more likely to be the manifestation of a regression error. In our preliminary evaluation of BERT, we have shown that the approach can identify subtle regression errors that may go otherwise undetected and result in failures in the field [2].

BERT was initially implemented as a prototype plug-in for the Eclipse IDE to support programs written in the Java programming language. To extend the approach to programs written in the C# programming language, we have successfully ported many features of BERT to the .NET platform as a Visual Studio 2010 plug-in. Like any other plug-in for analysis and testing of programs, the BERT plug-in needs support from the IDE to perform its tasks. Specifically, BERT must be able to access functionality for: intercepting save operations, triggering and checking build processes, computing program differences, generating test cases, instrumenting code, running test cases and code in general, visualizing information to the user, and allowing the user to interact with such information. This is a fairly rich and varied set of functionality, which makes BERT a good case study for discussing plug-in support within IDEs.

3. OUR EXPERIENCE

In this section, we discuss our experience in developing the BERT plug-in for two different platforms and IDEs. BERT's Eclipse plug-in is written in Java and uses several features provided by the Eclipse plug-in development framework [1] and by other external tools.

BERT uses built-in Eclipse features to intercept save and build events and invoke a build operation if it is not automatically invoked by the system. (Eclipse provides the user the option to enable and disable automated builds.) Regardless of whether the build is performed automatically or invoked by the plug-in, BERT checks its outcome. If the build is successful, BERT considers the newly generated version of the program as version V1, whereas the immediately previous version (i.e., version V1 of the previous invocation of BERT) is tagged as version V0 (see Section 2).

Given versions V0 and V1, BERT again leverages the IDE's functionality to compute the differences between V0 and V1 in terms of a list of changed classes between the two versions. For each changed class *c*, the plug-in uses a third-party tool, Randoop [5], to generate test cases for *c* in JUnit format (`http://www.junit.org/`). The resulting test cases and the code are instrumented using the bytecode rewriting library Javassist (`http://www.csg.is.titech.ac.jp/~chiba/javassist/`), so as to add probes that can log relevant state information at runtime. State information is logged in the form of XML files using the Java SAX library (`http://sax.sourceforge.net`). BERT runs the instrumented test cases on the instrumented versions V1 and V0 of the modified classes using Eclipse's support for JUnit and, after running the test cases, compares the dynamically collected state information to identify behavioral differences between the two versions.

Implementing BERT for Visual Studio required an almost complete rewriting of the code because Visual Studio plug-ins are written in the C# language. However, the modular design of the existing Eclipse plug-in helped us considerably, as we were able to reuse much of the code architecture and design. BERT's Visual-Studio plug-in also intercepts build events from the IDE, although the build mechanism is fairly different in the two environments, which required considerable changes in the implementation of the basic mechanisms. In addition, the differencing of builds is not directly supported by Visual Studio, so we had to develop and add to the plug-in a differencing engine. For test case generation, BERT leverages the PEX framework for unit test generation [8, 7], which is also available as a Visual Studio plug-in, together with a random method sequence generator. Instrumentation of the test cases generated by PEX and of the code is performed at the binary level using the CCI framework (`http://cciast.codeplex.com/`). Also in this case, the probes added to the code produce a log of relevant state information in XML format. Finally, similarly to its Eclipse counterpart, BERT for Visual Studio runs the instrumented test cases on the instrumented versions (old and new) of the modified classes, compares the runtime state information, and analyzes the results of the comparison to identify behavioral differences between versions V0 and V1 of the program.

As it can be inferred from the above discussion, the development of both BERT plug-ins was made easier—maybe even possible—by the support provided by Eclipse and Visual Studio (together with some third-part libraries and additional plug-ins). For instance, being able to automatically compute the differences between two versions within Eclipse was extremely useful, and so was being able to generate plug-in templates within Visual Studio. Nevertheless, our experience with the development of the two plug-ins made us realize that, as expected, neither of the two platforms provided all the functionality we needed to develop BERT. More generally, our experience allowed us to identify a set of important features that plug-in development frameworks should provide to support the integration of program analysis and testing techniques. We summarize our findings in the next section.

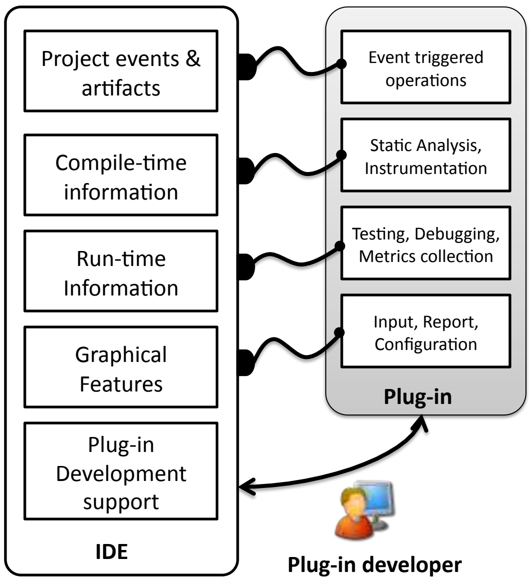
Project events & artifacts
Compile-time information
Run-time Information
Graphical Features
Plug-in Development support
IDE
Event triggered operations
Static Analysis, Instrumentation
Testing, Debugging, Metrics collection
Input, Report, Configuration
Plug-in
Plug-in developer

Figure 2: IDE's features for supporting plug-ins.

4. IDE SUPPORT FOR PROGRAM ANALYSIS AND TESTING PLUG-INS

Based on our experience developing BERT on two different IDE platforms, we identified some features that IDEs should support for the development of testing and analysis plug-ins. In this section, we discuss these features divided into several categories. To support the discussion, Figure 2 provides a graphical depiction of the features, and Table 1 summarizes them, along with information on how we used them and some concise comments on portability issues we encountered related to the features.

4.1 Project Artifacts and Events

Analysis and testing plug-ins work on a number of project artifacts, including code—either in source or compiled form—and configuration files. Therefore, such plug-ins should be able to easily access and modify these project artifacts. Plug-ins also need to be aware of software development activities, so that they can perform tasks that are dependent on a specific activity. Luckily, this aspect is usually well supported in existing IDEs, and plug-ins can be invoked in correspondence to a project activity through event notifications (e.g., when a user opens a file, move to a specific point in the code, compiles a project, runs a set of tests). This kind of event-driven programming allows plug-ins to perform their tasks right after the activity occurs.

4.2 Static Information

When accessing project artifacts, plug-ins must be able to retrieve different kinds of information. We list some examples of static information about a project that plug-ins may need to access.

Intermediate representations. Most IDEs perform at least basic syntactic checks on programs. To this end, they typically build Abstract Syntax Trees (ASTs) for the programs. Plug-ins can directly analyze such intermediate representations or, more typically, use them to derive other information, such as control- and data-flow graphs, for supporting more sophisticated analysis.

Program differencing. Plug-ins may need information on program differences to perform various kinds of analysis. For example, program-change information may be used to perform change analysis or, matched with dynamic information, perform regression testing tasks on evolving systems. IDEs connected to source control systems, such as CVS or SVN, can provide information about the change history. IDEs can also provide change information for two versions not yet committed to a repository.

Instrumentation. Plug-ins that perform dynamic analysis typically need to add probes (i.e., snippets of code) to a program, or even to modify certain statements. This type of code instrumentation is commonly used to perform tasks such as runtime monitoring, dynamic memory protection, or logging. (We consider instrumentation in this section because it operates on static artifacts.) IDEs may support instrumentation by providing access to internal information on the programs, such as ASTs, or by allowing a plug-in to freely access and modify a project's code (e.g., through the use of an external library).

Build parameters. Although this is a more mundane aspect, many analysis and testing plug-ins need to be able to read and modify the parameters passed to the compiler. For Java programs, for example, a plug-in might want to update the build path to load a third party library that is used by the instrumented code. Similarly, a plug-in might need to control a compiler option for getting access to dynamic information, as discussed in the next section. It is therefore important for an IDE to give access to these parameters to a plug-in through some programmatic interface.

4.3 Dynamic Information

Besides collecting static information about a project, plug-ins may also need to gather dynamic data, that is, data collected at runtime, while the program is executing. Similarly to what we did in Section 4.2, we provide examples of dynamic information that may be needed by a plug-in and of the functionality that a plug-in may require to gather such information.

Test execution. One common need for plug-ins that support testing related activities is the ability to execute test cases from within the IDE. In more general terms, plug-ins that collect any sort of dynamic information normally need to run the program whose information is being collected. This functionality is typically provided by modern IDEs, either through the integration of testing frameworks, such as JUnit or NUnit, or through specific mechanisms, such as application launchers in Eclipse.

Metrics. Obtaining runtime information from an execution is crucial for plug-ins that perform dynamic analysis. Metrics are generally collected with suitable instrumentation that is added to programs, as described earlier. Such metrics can be of several types, based on the data they collect. For example, *code coverage* computes which elements in the source code (e.g., statements, branches) are exercised by a given execution or set of executions. Coverage information can be used by plug-ins to report untested parts of the program or to perform further analysis (e.g. fault localization). Another example is *code profiling* data, which measure the number of times an event (e.g., the execution of a statement) occurred. IDEs often contain profiling tools that provide such information.

Debugging information. Plug-ins may also need to interface with a debugger to assist debugging activities by providing more information to developers. This information may, for instance, include details from the current and past executions to give developers more insight about the program's behavior.

Table 1: Features needed to support testing and analysis plug-ins, how we used them, and comments on their portability.

IDE feature	Usage in BERT	Portability comments
Project Artifacts and Events	Capture save and build events, access build files.	Event notification and access to project artifacts are well supported, although in very different ways, in both IDEs.
Static Information	Find program differences, instrument code.	Both IDEs provide limited information and must typically be complemented by third-party libraries and tools.
Dynamic Information	Execute automated tests, collect runtime metrics.	Both IDEs provide a mature infrastructure for test execution. Metrics collection is mainly dependent on static instrumentation performed using third-party libraries.
Graphical Features	Input configuration, show results to the user.	The two IDEs have different GUI-interaction mechanisms, but suitably defined data formats can be reused.
Plug-in Development Support	Use development and debugging features.	The two IDEs provide similar plug-in development environments and features. Documentation on APIs is often outdated or incomplete in both IDEs.

4.4 Graphical Features

Plug-ins must interact with users to receive inputs and provide them with information they generate. Extensible IDEs typically support this types of interactions in several ways. In particular, IDEs provide mechanisms to trigger a plug-in's functionality from different contexts. (For example, a plug-in's feature can be launched from an object's contextual menu.) And the output of a plug-in can typically be presented using different views provided by the IDE. These views include, among others, in-editor highlighting, information dialogs and windows.

4.5 Plug-in Development Support

Development. Since plug-ins are also software, they are developed just like any other software project within the IDE. To successfully implement a plug-in, developers need to be familiar with IDE features and APIs that the plug-in can leverage. Because these APIs can be extensive, it is essential for the API documentation of a plug-in development framework to be complete, clear, well organized, and easily accessible. Sample code and informal documentation in the form of blogs also help developers understand and use the IDE features in their plug-ins. However, such external documentation tends to be less rigorous and can easily become outdated, so it cannot replace the information found in the official documentation of the IDE. If such documentation is not available, or difficult to access and consume, developers may re-implement features that already exist or unnecessarily include third-party libraries, which can result in code that is more complex and, more generally, in wasting resources. When documentation is missing or incomplete, another (more subtle) issue is that developers may mistakenly use unpublished internal APIs. This practice, which is unfortunately fairly common, very often results in failures of the plug-in after an update of the IDE or its libraries.

Testing, Debugging, and Maintenance. To help developers test, debug, and maintain their plug-ins, both Eclipse and Visual Studio provide the possibility of launching a separate test instance of the IDE that loads the plug-in and can be monitored from the primary IDE instance. This is essential during plug-in development, as it helps developers exercise their plug-ins without having to go through the lengthy process of deploying and installing them.

5. CONCLUSION

Modern IDEs have evolved into complex environments that provide an increasingly large set of features to support most aspects of software development. In addition to their core features, IDEs also provide developers with the possibility of extending such features through the development of plug-ins. In this paper, we focus on the support provided by modern IDEs to the development of program-analysis and software-testing plug-ins. To this end, we start from our experience with the development of a specific plug-in that supports automated regression testing for two IDEs: Eclipse and Visual Studio. (We developed the plug-in first for Eclipse and later ported it, with some significant modifications, to Visual Studio.)

We then identify, based on our experience, a set of characteristics that should be provided by an IDE to suitably support testing and analysis plug-ins. This set of characteristics includes, among others, the ability of plug-ins to access various project artifacts, gather different kinds of static and dynamic information, and leverage graphical capabilities of the IDE.

As we also stressed in the paper, our goal is not to be comprehensive, which would not be possible given the space available, but rather to provide a starting point for a discussion on this topic and on the more general issue of plug-in support within modern IDEs.

6. REFERENCES

[1] E. Clayberg and D. Rubel. *Eclipse: Building Commercial-Quality Plug-ins (2nd Edition) (Eclipse)*. Addison-Wesley Professional, 2006.

[2] W. Jin, A. Orso, and T. Xie. Automated behavioral regression testing. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation (ICST 2010)*, ICST '10, pages 137–146, Washington, DC, USA, 2010. IEEE Computer Society.

[3] W. Jin, A. Orso, and T. Xie. Bert: a tool for behavioral regression testing. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2010)*, pages 361–362, New York, NY, USA, 2010. ACM.

[4] Microsoft Corporation. Microsoft visual studio 2010 - the official site of visual studio 2010. `http://www.microsoft.com/visualstudio/en-us`, Jan. 2011.

[5] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, OOPSLA '07, pages 815–816, New York, NY, USA, 2007. ACM.

[6] The Eclipse Foundation. Eclipse - the eclipse foundation open source community website. `http://www.eclipse.org/`, Jan. 2011.

[7] Pex and Moles - Isolation and White box Unit Testing for .NET. `http://research.microsoft.com/en-us/projects/pex/`, 2011.

[8] N. Tillmann and J. De Halleux. Pex: white box test generation for .net. In *Proceedings of the 2nd International Conference on Tests and Proofs (TAP 2008)*, pages 134–153, 2008.