Integrating a Set of Contract Checking Tools into Visual Studio

Manuel Fähndrich, Michael Barnett, Daan Leijen, Francesco Logozzo
Microsoft Research
Redmond, USA

***Abstract*—Integrating tools and extensions into existing languages, compilers, debuggers, and IDEs can be difficult, work-intensive, and often results in a one-off integration. In this paper, we report on our experience of building and integrating the CodeContract tool set into an existing programming environment. The CodeContract tools enable 1) authoring of contracts (preconditions, postconditions, and object invariants), 2) instrumenting contract checks into code, 3) statically checking code against contracts, and 4) visualizing contracts and results. We identify three characteristics of our integration that allowed us to reuse existing compilers and IDEs, increase the reach of our tools to multiple languages and target platforms, and maintain the tools over three consecutive versions of C# and Visual Studio with little effort. These principles are 1) use source embedding for new language features, 2) use target analysis and rewriting, and 3) use generic plug-ins to isolate tools from the IDE.**

***Keywords*-tools; plug-ins; contracts; IDE**

I. Introduction

Programming language researchers and professional tool writers need to get their language extensions and analysis tools into the hands of professional developers in order to have impact and learn from user experience. On the other side, the professional programmer requires tools to provide 1) easy adoption, 2) immediate benefit, and 3) low risk.

Teams consist of many developers with established development, build, and test practices. *Easy adoption* of a tool means that these practices are impacted minimally. This requirement leaves tool writers with little space in which to deviate from existing practice. In particular, developers typically don't have the luxury to switch languages or programming environments in order to adopt a new tool.

Developers need to see *immediate benefit* from using a new tool, otherwise, it easily falls by the wayside. It may be beneficial to provide a combination of tools that provide some small immediate low cost benefit, with the promise of higher long-term benefit at a higher cost.

Tool writers must attempt to provide *low risk* to teams adopting the tools. Tools often are buggy; what is the risk of abandoning the tools when a dead-end is reached? Will it be necessary to change a lot of code or practice, leading to further cost and delays? If the tools do code generation, what's the risk of generating bad code that will only be found after shipping? How easy is it to mitigate this issue?

In this paper, we report on our experience of building and integrating the CodeContract tools [1] into an existing programming environment. We discuss how we achieved the above mentioned goals of easy adoption, immediate benefit, and low risk, and how these affected our design and the integration with the host programming environment, in our case, Visual Studio.

The main lessons we draw from our experience for integrating into an existing development environment are:

Source Embedding: Embed new language features into an existing source language using only existing language features such as attributes and calls to special libraries, rather than new syntax or stylized comments.

Target Rewriting: Give semantics to the new features by rewriting the compiler output (the target language), rather than the source input. Similarly, perform analysis of code at the target language level, rather than the source level.

Generic Plug-ins: For IDE integration, write plug-ins that are reusable across many tools rather than a single specific one. This means that generic plug-ins are themselves pluggable with tool specific extensions. The main two generic plug-ins we wrote are 1) a property and settings manager to visualize, edit, and persist tool specific settings into whatever format the development environment uses to store such info about a build unit, and 2) a feedback manager that translates tool output such as warnings and errors into IDE specific warning lists and source squigglies.

These design decisions have served us well and have provided our tools with a great amount of *leverage*:

- The source embedding allows using our language features in both C# and VisualBasic without a single line of change to the existing compilers and IDE.
- Our new features always work with the latest version of these languages. During the lifetime of CodeContracts, the compilers and languages have changed twice already (v3.5, v4.0, v4.5).
- Analyzing and rewriting the target instead of the source makes our tools language agnostic. The same tools work on C# and VisualBasic output.
- The use of generic plug-ins isolates our extensions from changes in the underlying IDE. We are on our third version of Visual Studio (2008, 2010, and now 2012) with no change to the tools and a few minor changes to the installers.
- The loose integration of our tools at the level of the source language and the .NET target language make our tools usable for a variety of platforms supported by

Visual Studio, namely desktop (VB and C#), Silverlight inside and outside of browser (VB and C#), Windows Phone Applications (VB and C#), and server side ASP.NET and Azure code (VB and C#).

The rest of the paper is organized as follows: Section II provides some background on the contract tools, Section III discusses the principles of embedding and target rewriting we employed, Section IV explains the MSBuild process and the hooks in Visual Studio necessary for our approach. Section V discusses our generic plug-ins that isolate our tools from the Visual Studio IDE and Section VI recaps our lessons learned.

II. CodeContracts

We use the CodeContracts tools [2], [3] as the poster child to illustrate our preferred methodology for language extension and tool integration. CodeContracts enable 1) authoring of contracts (preconditions, postconditions, and object invariants), 2) instrumenting contract checks into code, 3) statically checking code against contracts, and 4) visualizing contracts and results. Upon embarking on this project, we gave ourselves a firm constraint that the entire tool chain and experience must integrate into an existing programming environment, in our case, Visual Studio, benefiting developers who do not have the luxury to rely on research compilers. We carefully picked our extension and interaction points in order to 1) minimize the integration work, 2) make our tools usable on multiple languages and target platforms, and 3) keep the tools working from one version of Visual Studio to the next.

An example of CodeContracts and its embedding in C# is shown in Fig. 1. The example illustrates the source embedding approach. We created a library called Microsoft.Contracts.dll which contains a class Contract with a number of static methods: Requires, Ensures, etc. These methods return nothing and take a single boolean argument. In this example, we use calls to these methods to "declare" preconditions and postconditions. The C# compiler treats them as ordinary method calls: it typechecks the arguments and emits MSIL [4], an object oriented bytecode, which evaluates the arguments and calls the methods. Such code is not useful to run directly, but our tools extract the MSIL that evaluates the expressions and the calls from the target and use them to perform instrumentation, documentation generation, or static analysis.

It is worth noting that the use of a library does not preclude later making the special classes and methods of an embedding approach a more integrated feature. For example, in version v4.0 of the .NET Common Language Runtime (CLR), the Contract class and methods were integrated into the basic class library. Thus, starting with v4.0, the external library was no longer necessary. The library can still be used however to build for an older version of the CLR, as our tools do not require the special classes to be in a particular library.

```
string Compute(string str, int index, Collection c, out int len)
{
  Contract.Requires(str == null ||
                    0 <= index && index < str.Length);
  Contract.Ensures(str == null ||
                   ! String.IsNullOrEmpty(Contract.Result())
                   && c.Count > Contract.OldValue(c.Count));
  Contract.Ensures(Contract.ValueAtReturn(out len) >= 0 );
  Contract.Ensures(str == null ||
                   Contract.ForAll(
                     0, Contract.ValueAtReturn(out len),
                     i => Contract.Result()[i] == s[i])  );
  ...
}
```

Figure 1. Example of an embedded language feature: CodeContracts

III. Embedding and Target Rewriting

Programming language extensions have used two main approaches in the past: 1) entirely new programming languages or syntactic extensions of existing ones, or 2) stylized comments in existing languages. In either case, these approaches require entire compiler infrastructures to support tools acting on the new language features. Specialized languages are difficult to get into general usage, as the compilers and support tools are usually not on par with commercial product quality. Often such infrastructures need to track the evolution of some original language (e.g., Spec# [5] vs. C# and JML [6] vs. Java), which means they either don't support the same language, or lag several years behind the features of the main language.

To side-step all these issues, we advocate language extensions via an *embedding* [2] approach. The idea of embedding a new feature in an existing programming language is to:

1) express the new feature as statements in the existing language itself consisting of calls to a special library,
2) leverage the existing language compiler to perform name and overloading resolution, type checking, and code generation, and to
3) extract the use of the new language feature from the compiled target code using decompilation to find the calls (and arguments) to the special library.

The embedded approach for new language features provides numerous benefits to the programmer:

- Not only can the existing editor and IDE be used to author the new features, but the IDE actively supports writing proper expressions by providing highlighting, completion, intellisense, and early feedback on erroneous expressions (due to the fact that the existing language will background check the expressions as normal code).
- Refactoring tools work properly on the new features as well, e.g., renaming a parameter will rename any parameter use inside a new feature as well. Contrast

this to having new features implemented via attributes or special comments in code.

Thus, developers don't have to learn a new language, a new compiler, or a new IDE. Embedding is also beneficial to writers of tools:

- Since the features are compiled by the existing compiler, the tool writer has no need to duplicate the full compiler infrastructure, such as the parser, type checker, name and overloading resolution, etc., nor extend the IDE to recognize the new constructs.
- Extracting the new feature use from the compiled target code as opposed to the source code allows the tool writer to deal with a smaller and usually better specified language than the original source language. In our example, consider the difference in complexity between the full C# language and the relative simplicity of the target MSIL intermediate language of .NET. Additionally the new features may work on other languages that compile to the same target (in our case VisualBasic).
- The tool writer can typically reuse existing well tested infrastructure to manipulate/analyze the target code, such as .NET binary reader/writers, or similarly Java byte code infrastructures.

Risk Mitigation

In the introduction, we alluded to the need to minimize risk for adopters of new tools. In particular, it should be possible for adopters to stop using the tools without negative impact to the project (beyond the lack of the benefit of the tool itself). In particular, this implies that dropping the tools should not require any code changes.

With source embedding of new features it is thus a good idea to employ techniques provided by the compiler or preprocessor to make sure that all new feature use can be *erased* by the compiler via suitable compiler options. In our example of CodeContracts, we use *conditional attributes* on our static methods[1]. These attributes act as an implicit #ifdef CONTRACTS ... #endif bracketing around every *use* of our contract methods. As a result, if the CONTRACTS symbol is not defined during a build, the code compiles as if all extension usage had been syntactically erased. This principle of erasure gives adopting teams a lot of confidence in trying out new features and tools, since they know they can flip the switch at any time without negative impact.

The same principle also allows teams to use new features and tools for debugging only, but ship code that contains no use of new features and requires no target rewriting. Removing new tools from the critical path from source to binaries mitigates risk and puts adopters at ease.

IV. Visual Studio and MSBuild

Visual Studio is a full-fledged integrated development environment from Microsoft. It supports various source languages and target platforms, has language specific editors with high-lighting and intellisense, and all the usual bells and whistles one expects from a modern IDE today.

[1]Conditional attributes are a standard C# and VisualBasic feature.

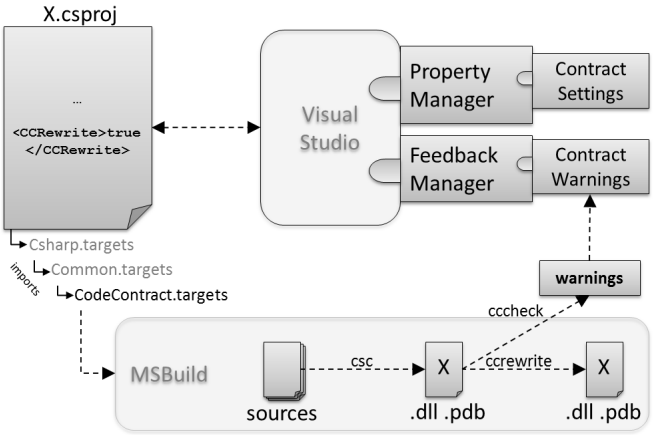

Figure 2. Architecture

For C# and VisualBasic, code is grouped into *projects*, where each project consists of a number of source files and results in a single *managed assembly* containing the metadata and IL code. These assemblies typically reside in .dll or .exe files. Debugging information for assemblies is stored in separate .pdb files.

Project information is stored in project files (.csproj for C# and .vbproj for VisualBasic). These files contain XML conforming to MSBuild [7] descriptions. MSBuild descriptions are similar to the classic Unix Makefiles, and the msbuild command corresponds to the classic make command [8]. Project files define source files, build flavor, references to libraries, target names etc. as *properties*. The project files don't contain any actual build rules. Build rules are factored separately in shared files such as CSharp.targets, VisualBasic.targets, and Common.targets which are included from every project file. These files also contain XML conforming to MSBuild descriptions. In this case, they contain definitions of build steps that are parameterized by properties defined in the project files. Fig. 2 illustrates this situation for a C# project called X. The project file X.csproj is used by MSBuild to create X.dll from the sources using the C# compiler csc. The project file imports the general C# build rules via CSharp.targets, which in turn include general build rules (for both VisualBasic and C#) from Common.targets.

MSBuild is used to build project outputs from the command line as well as invoked by Visual Studio when a build is triggered via the IDE. Visual Studio provides a graphical user interface that exposes the standard project properties of project files. Therefore, the way Visual Studio controls the build works entirely by it changing the properties in the project files and letting MSBuild do the build based on these properties alone.

Hooks

To influence the build process, the Common.targets provides a general hook to import additional MSBuild files. The hook imports all MSBuild files located in a particular configuration directory on disk. The Common.targets build rules are written in such a way as to make it easy to write additional rules that trigger during a build prior or after certain other build steps. Therefore, extending and modifying the standard build requires no changes to any existing build files. Instead, one simply authors an additional MSBuild description containing new rules and settings and imports it using the existing build hook.

Figure 2 shows how this hook is used to import the Contracts.targets file. It defines how to rewrite project outputs to instrument contracts or to run the static analysis. In our example, it adds an extra build step after the compilation via the C# compiler csc to invoke the contract rewriter ccrewrite to instrument checks into the target IL of X.dll.

The approach of using the existing build hook provided for all project types makes our approach independent of project type and addresses some of our overall goals as follows: the alternative approach (no common build hook) would be to define a new kind of project type (C# with contracts, and similarly for VB and other flavors) and create new projects based on this type. The new project type can have its own build rules and would not require an existing hook. However, it would a) fail the easy adoption test, since developers would not be able to use the tools on *existing* projects, b) entail higher risk, as a development team would have to change all their projects to switch back to not using the tools. With our advocated approach, it is even possible for some team members of a development team to use the tools, and for other to not install them. The build works in both scenarios, one with tools, the other without.

V. Plug-in Architecture

With the build hook described in the previous section, we can already influence the build in order to run extra tools. In principle, we don't need any additional integration into the IDE. E.g., to perform the contract instrumentation step, the project files simply need to set the property <CCRewrite> to true and the build files handle the rest. Similarly, output from the tools such as warnings are already printed as part of the build log.

Additional integration into the IDE is really only needed to make the tools more accessible. Normally, developers don't edit the XML in project files by hand, nor do they look at the msbuild output log to see errors. Instead, the IDE provides a graphical user interface to read and change project settings and synchronizes them with the corresponding properties in the project files. Similarly, the IDE takes care of nicely displaying warnings and errors in a sortable list, and additionally may overlay squigglies or context menus over the source code.

Thus, in order to make our tools more accessible, we wrote two generic plug-ins for Visual Studio that we describe in the next sections.

A. Property Manager

The property manager is a generic plug-in written as a Visual Studio package. It provides an interface for additional plug-ins to read and write tool specific project properties into existing project files of C# and VisualBasic. Specific plug-ins into the property manager consist of a UI component typically showing check marks and other setting elements. In our example, the contract settings plug-in shown in Figure 2 on the top-right plugs into the generic property manager to read contract specific settings from the project file, display them, and to write changes from user manipulations back to the project file via the property manager.

The property manager provides an important level of isolation and abstraction from the underlying IDE. Implementing the property manager was quite difficult. To properly interface with Visual Studio and display settings on existing project types (such as C# or VisualBasic) required a rather deep integration, which is beyond the casual plug-in writer. The property manager thus encapsulates this complexity and makes it possible to write many tool specific settings plug-ins in a very easy way, well within the reach of anyone who can write some simple C# or VisualBasic code. Additionally, the property manager isolates tool specific plug-ins from changes in the underlying IDE due to new versions.

Our plugin for CodeContract properties works for both C# and VisualBasic. In fact, the same component is used for both. The CodeContract properties show up after all standard C#/VB project properties as shown in Fig. 3.

B. Feedback Manager

The feedback manager is our second generic plug-in written as a Visual Studio package. It provides additional plug-ins an interface to the error list maintained by the IDE, to source code overlays (such as squigglies to underline parts of code with warnings), and context menus on warnings and squigglies. The feedback manager again is generic in that it takes care of the complicated integration with Visual Studio once and for all, and provides a simple interface to plug-ins such as the one we wrote for CodeContracts. The contract specific feedback plug-in uses context menus to display related warning locations (such as the location of the original precondition violated at a call site).

Additionally, the contract specific plug-in into the feedback manager allows background execution of the static analysis (to avoid slowing down the build). The same benefits of abstraction and isolation discussed for the property manager apply equally to the feedback manager as well.

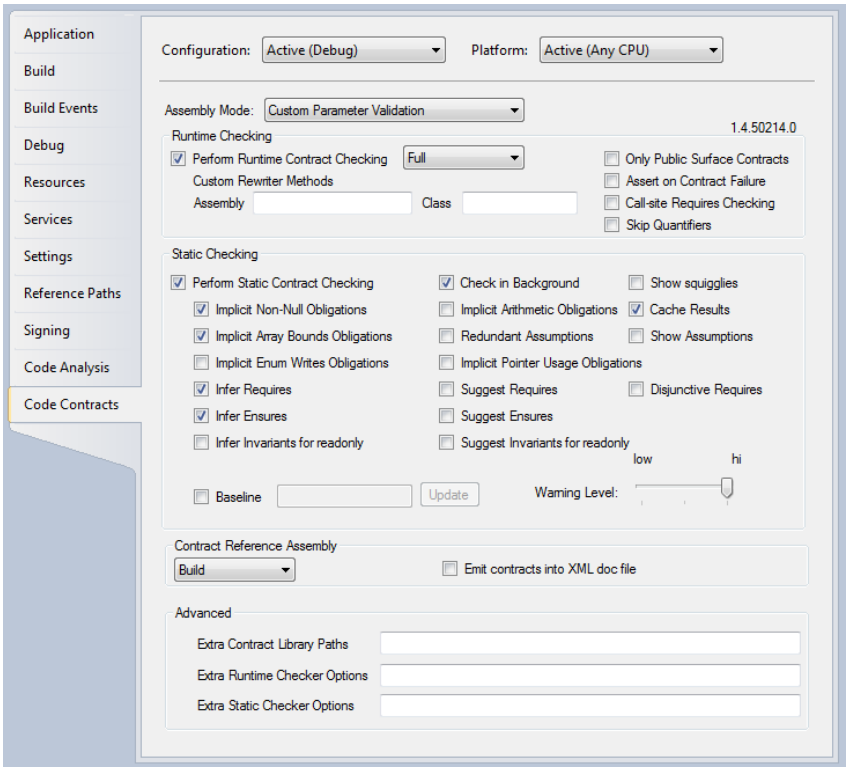

Figure 3. CodeContract Property User Interface

VI. Lessons Learned

In this section, we reflect on our approach and discuss what worked and why, and point out where our approach has short-comings.

Source embedding of new features and target rewriting worked really well for tracking language changes and supporting the new features with the latest language versions. The main reason for this advantage is that the target language typically evolves much slower than the source language. This was the case for .NET vs. C# and VisualBasic. E.g., in v4.0, C# added variance for generics, dynamic types, named arguments and optional parameters, among other features, whereas the .NET IL didn't change at all.

The reason why our build integration works well and is stable is thanks to a good existing design and extensibility of the common MSBuild rules in the shared Common.targets.

The languages supported by our approach are not really dependent on the languages per se, but more by how they integrate into Visual Studio and the build, and what target binary language they support. VisualBasic and C# are integrated very similarly into Visual Studio and thus our tools work the same on both. C++ and F# on the other hand have enough differences that the integration does not work. For C++ it is the binary target language that is not .NET and the build process is completely different. For F#, the differences are small and our approach should work barring a few technical difficulties.

Due to our reuse of minimal custom integration, our tools work for a surprising variety of .NET platform flavors. There are Silverlight applications running in the browser, or stand-alone, there are ASP.NET web sites and Azure services with server side .NET code, and Windows Phone applications. All these platforms have slight differences in library support, but they share the same source languages, target language, and build structure. Our tools work with all of these without change[2].

The integration work we did for CodeContracts can be reused in future tools as most of our integrations are non-tool specific and use existing hooks provided by Visual Studio. In fact, we already have used essentially the same approach for other tools, such as the concurrent revision rewriter [9].

Our approach also has short-comings. Target analysis and rewriting is at the mercy of the precision of debugging information in .pdb files. E.g., for .NET source mappings from target IL to source is based on lines only, so precise intra-line information is not available when highlighting an expression with a warning. Target rewriting also has the disadvantage of obscuring high-level constructs in the source language. E.g., iterators are compiled away into a number of helper classes and methods. Some decompilation must be performed in order to extract contracts from such methods.

Another problem we ran into with the build hooks is

[2]The only platform specific code is in selecting the appropriate contract reference assemblies for the platform during the build.

that they don't provide any ordering/scheduling support for multiple rewriters in the tool chain. This makes it difficult to combine tools that don't know about each other.

VII. Related Work

The most comparable effort to the CodeContracts tool set and its IDE integration are the various tools developed for the Java Modeling Language JML [6]. JML uses a comment-based syntax to augment Java programs with pre and post conditions, and object invariants. JML has a long history of tools for runtime checking, static checking, and documentation generation, starting in 1999. As expounded in [10], there are at least five distinct efforts and versions of similar JML tools [11], [12]. All these tools seem to be based on modified Java compilers that augment various phases to parse the JML expressions and either translate them to runtime checks, or to various static checking backends. Some tools have various levels of Eclipse IDE [13], [14] support.

The JML approach has some advantages over our source embedding approach: the syntax for contracts can be more concise due to the ability to step outside the underlying language expression syntax (e.g., for referring to the result and old-values). Furthermore, we are relying on conditional compilation features to force erasure of contracts on builds where they are not desired, comment-based approaches obviously do not require such features as comments are erased automatically.

The amount of effort that went into these numerous open-source and community maintained tools over the years seems disproportionately larger than our own efforts, and yet has not resulted in a stable set of tools that has remained up-to-date with the Java language. In contrast, our approach and design has not changed since its beginning in 2007. Our tools have been completely built and maintained by three project members and two additional researchers, all on a part time basis. Most of our work has been spent on the actual tooling, such as the static checker engine, and the rewriting, and very little effort overall went into the build and IDE integration and its maintenance.

VIII. Conclusion

We identified three characteristics that stand out in our effort to build and integrate a set of contract checking tools into Visual Studio. 1) Use source embedding for new language features in order to reuse existing compilers and source editors as-is, 2) use target analysis and rewriting to isolate from the evolution of source languages and to support multiple source languages with a common set of tools, and 3) use generic plug-ins to isolate tools from the IDE.

Our tools and integration have survived three consecutive versions of the C# and VisualBasic languages and compilers, as well as three versions of Visual Studio with little maintenance effort on our part.

If we were to embark on another language extension project, we would proceed with the same design.

Acknowledgment

The authors would like to thank Herman Venter for his tireless efforts building the best .NET readers and writers.

References

[1] M. Fähndrich, M. Barnett, and F. Logozzo, "CodeContract Tools," Mar. 2009. [Online]. Available: http://research.microsoft.com/contracts

[2] M. Fähndrich, M. Barnett, and F. Logozzo, "Embedded Contract Languages," in *ACM Symposium on Applied Computing*, 2010.

[3] M. Fähndrich and F. Logozzo, "Static contract checking with abstract interpretation," in *FoVeOOS*, 2010, pp. 10–30.

[4] ECMA, "Standard ECMA-355, Common Language Infrastructure," http://www.ecma-international.org/publications/standards/Ecma-335.htm, Jun. 2006.

[5] M. Barnett, K. R. M. Leino, and W. Schulte, "The Spec# programming system: An overview." in *CASSIS*, ser. LNCS, vol. 3362. Springer, 2004.

[6] G. T. Leavens, A. L. Baker, and C. Ruby, "Preliminary design of JML: A behavioral interface specification language for Java," *SIGSOFT*, vol. 31, no. 3, pp. 1–38, Mar. 2006. [Online]. Available: http://doi.acm.org/10.1145/1127878.1127884

[7] Microsoft, "Standard ECMA-355, Common Language Infrastructure," http://msdn.microsoft.com/en-us/library/0k6kkbsd.aspx, Jun. 2006.

[8] S. Feldman, "Unix make tool," http://en.wikipedia.org/wiki/Make_(Unix).

[9] S. Burckhardt, A. Baldassin, and D. Leijen, "Concurrent programming with revisions and isolation types," in *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2010.

[10] D. Cok, "OpenJML: JML for Java 7 by Extending OpenJDK," in *NASA Formal Methods*, ser. LNCS, vol. 6617. Springer, Mar. 2011, pp. 472–479.

[11] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An overview of JML tools and applications," in *International Journal on Software Tools for Technology Transfer*, vol. 7, no. 3. Springer, Jun. 2005, pp. 212–232.

[12] A. Sarcar and Y. Cheon, "A new eclipse-based jml compiler built using ast merging," University of Texas at El Paso, Tech. Rep. TR #10-08, 2010.

[13] "Eclipse," `http://eclipse.org`, 2011.

[14] J. McAffer and J.-M. Lemieux, *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java(TM) Applications*. Addison-Wesley Professional, 2005.