Playing Cupid: The IDE as a Matchmaker for Plug-Ins

Todd W. Schiller and Brandon Lucia

Department of Computer Science
University of Washington
Seattle, Washington
{*tws,blucia0a*}*@cs.washington.edu*

***Abstract*—We describe a composable, data-driven, plug-in ecosystem for IDEs. Inspired by Unix's and Windows PowerShell's pipeline communication models, each plug-in declares data-driven capabilities. Developers can then seamlessly mix, match, and combine plug-in capabilities to produce new insight, without modifying the plug-ins.**

We formalize the architecture using the polymorphic lambda calculus, with special types for source and source locations; the type system prevents nonsensical plug-in combinations, and helps to inform the design of new tools and plug-ins. To illustrate the power of the formalism, we describe several synergies between existing plug-ins (and tools) made possible by the ecosystem.

I. Introduction

IDEs and plug-ins that provide apt information and action suggestions greatly boost developer productivity. For example, consider a developer who has two plug-ins installed: `RefactorSuggest`, a plug-in that suggests and performs refactorings, and `PerformancePredict`, a plug-in that estimates the performance of the program. Being performance-conscious, the developer can run `PerformancePredict` after performing a refactoring with `RefactorSuggest` to ensure the modification hasn't significantly degraded performance — the developer is manually filtering refactoring suggestions. In a perfect world, the filtering would be performed automatically; refactorings which harmed performance would not be suggested so that the developer would have the benefits of refactoring suggestions without having to worry about performance.

But, the developer isn't just concerned with her local copy, she also cares about the performance of the program that is created when she merges her local changes with the repository using the plug-in `VersionControl`. So, she must invoke `PerformancePredict` on the merged copy before committing. At commit time, however, it may be onerous to rollback the refactoring that degraded performance; the developer should have known about the performance degradation at the time of the refactoring. In a perfect world, the refactorings that would harm performance when merged might not even be suggested.

Modern IDEs like Eclipse can handle both filtering scenarios. In the first case, the author of `RefactoringSuggest` can declare an extension point, and the author of `PerformancePredict` can then modify their plug-in to contribute to the extension point. Adding `VersionControl` is also possible: the author of `PerformancePredict` could define an extension point for plug-ins that modify source code, and the author of `VersionControl` could then modify their plug-in to contribute to the extension point. In practice, this is neither realistic nor desirable. The author of `PerformancePredict` cannot be expected to define such an extension point which is not related to the functionality of the plug-in. Additionally, the author of `VersionControl` should not be expected to choose to contribute specifically to the plug-in `PerformancePredict`, lest they be expected to contribute to any and every Eclipse plug-in.

This is discouraging given the natural fit between the capabilities of the plug-ins: `RefactorSuggest` suggests changes to the program source, `VersionControl` produces changes to the program source by performing a merge, and `PerformancePredict` analyzes source code.

In this paper we present a plug-in model for IDEs that captures this opportunity for communication between plug-ins, even those plug-ins that are independently developed. Based on this model, the *end-user* can define filters, and create pipelines of analyses to produce novel insight.

A model of IDEs: IDEs are predominately modal, providing modes for development, testing, debugging, and synchronization. Within these modes, analyses are performed which can be displayed to the user in views, or used to determine which actions are available to the user. A plug-in provides some additional set of modes, analyses, views, and actions to the IDE platform, or other plug-ins, by contributing to explicit extension points.

Table I shows a subset of the modes, analyses, views, and actions provided by Eclipse, and some plug-ins. As our opening example suggests, new insight can be gained when views, analyses, and (possibly) actions are mixed between modes. However, remarkably little work has been done in this space, especially within the IDE.

IDEs as matchmakers: We describe a composable plug-in ecosystem in which the IDE plays matchmaker for data-driven plug-in capabilities. Each plug-in publishes its capa-

Table I
A SUBSET OF THE MODES, ANALYSES, VIEWS, AND ACTIONS PROVIDED BY ECLIPSE, AND SOME PLUG-INS. ENTRIES IN ITALICS ARE POSSIBILITIES THAT COULD BE REALIZED USING THE DESCRIBED PLUG-IN ECOSYSTEM. EACH OF THESE IS DESCRIBED IN SECTION III.

Eclipse Features			
Mode	Analyses	View	Actions
Development	Compilation, *Code Clone Detection §III-D*	Compilation Errors, JavaDoc, Class Outline, TODOs, *Unit Test Results §III-C*, *Merge Conflicts §III-A*, *Code Clones §III-D*	QuickFix, Refactor, Format
Debugging	Evaluate Expression	Watch, Stack Trace, *Test Coverage §III-E*, *Code Churn §III-E*	Set / Remove Breakpoint Step / Pause / Continue
Testing	Run Tests, Code Coverage	Unit Test Results, Code Coverage	Run Test(s)
Synchronization	Conflicts, Diff	Synchronize View, Diff, Log	Commit, Revert, Update
Verification (1)	Extended Static Checking *Daikon Contract Inference §III-B2* *Dynamic Contract Violations §III-B1*	Contract Warnings, Contract Suggestions	Check

bilities to a "bulletin board" in the IDE, allowing other plug-ins to use the capability. End-users can then seamlessly mix, match, and combine capabilities to produce new insights, *without modifying the plug-ins.*

Our model is inspired by Unix's data pipeline and Windows PowerShell's object pipeline. These systems are made up of small, single-purpose, programs that communicate via text and objects, respectively. This paper makes three main contributions:

1) A formulation of plug-in capabilities in the polymorphic lambda calculus (2) that describes how plug-ins can communicate; the formalism has the added benefit of suggesting new analyses that can be invented by combining existing capabilities.
2) Filters, which allow the *end-user* to filter out extraneous analysis or undesirable action suggestions (e.g., bad Quick Fix (3) suggestions).
3) Pipelines, which allow the *end-user* to combine data-driven plug-in capabilities to produce novel insight.

To explore potential interactions between plug-ins, we describe novel analyses and views that the end-user could create using our system. Many of these are created by allowing users to view the results of an analysis outside of that analysis's "natural" mode, e.g., showing code churn when debugging.

In Section II we formalize the proposed plug-in ecosystem by introducing the types, filters, and pipes. In Section III, we describe example synergies between existing plug-ins (and tools) that can be captured using the model. In Section IV, we discuss the limitations of the model and extensions to the model to address those limitations. In Section V we describe related work in pipeline programming and mash-ups. We conclude in Section VI.

II. PLUG-IN ECOSYSTEM

In this section, we describe the design space for the data-driven IDE plug-in ecosystem. When the IDE loads, each plug-in publishes the type of capabilities it provides to the IDE's bulletin board (IDE components also publish capabilities, as if they were plug-ins). Similarly, when the user creates a filter or capability pipeline, a description of its type is published to the bulletin board.

A. Basic Types

To ensure that the data being communicated between plug-ins is sensible, we formalize plug-in capabilities using types in a typed lambda calculus extended with pairs, type lists, and type sets:

```
t ::= bool | number | string |
      source | location |
      t -> t | (t, t) | {t*} | [t*]
```

Here $source$ is the type for the whole source of a project. For a Java project, this includes source files, build configuration files, etc. The $location$ type refers to a location in the source. Given these types, a plug-in can publish two basic interfaces:

- Program Transformation: $source \rightarrow source$
- Program Analysis: $source \rightarrow \tau$

, where τ is any of the aforementioned types. Two useful type *aliases* are:

- $action \equiv source \rightarrow source$
- $label \equiv (location, \tau)$, a source annotation

These aliases allow the results of common analyses to be described naturally and succinctly:

- Program trace: $[label^*]$, where each label annotates a single instruction
- Test suite results: $\{label^*\}$, where each label is a test and its outcome
- Call graph: $\{(location, location)^*\}$, a set of call graph edges
- Action Suggestion: $(source, action)$

Consider the example from the introduction, with plug-ins `RefactorSuggest`, `PerformancePredict`, and `VersionControl`. The plug-ins would publish the following capabilities:

Published Capabilities (from Example)	
`RefactorSuggest`	$source \rightarrow \{(source, action)^*\}$
`PerformancePredict`	$source \rightarrow number$
`VersionControl` (Merge)	$action$

In words, `RefactorSuggest` is a program analysis that produces action suggestions, `PerformancePredict` is a program analysis that produces a single number (a score), and `VersionControl` is a source transformation (action). In Sections II-C and II-D, we introduce a few pieces of additional machinery to combine the plug-in capabilities in order to hide refactoring suggestions that would harm the performance of the program when merged.

B. Parametric Types and Bounded Polymorphism

To further prevent nonsensical plug-in matching, additional types can be introduced for source and source locations:

```
source ::= text | java | bytecode |
           trace
location ::= text | method | test
```

To aide exposition, we'll write the source and location types *as if* they parametrized the source and location types:

- $source{<}L{>}$, where L is the language of the source code
- $location{<}T{>}$, where T is the type of location

We'll use the same notation for labels:

- $label{<}T, A{>}$, where T is the location type and A is the type of annotation

With such types, a Java compiler can be viewed as offering both a checking and compilation capability:

Java Compiler Capabilities	
Check	$source{<}java{>} \rightarrow \{label{<}text, string{>}^*\}$
Compile	$source{<}java{>} \rightarrow source{<}bytecode{>}$

Bounded polymorphism à la System $F_{<:}$ (2) can be used to further clarify the type of a capability. For example, the `VersionControl` capability has the type: $\forall L <: text.\ (source{<}L{>} \rightarrow source{<}L{>})$ since it can merge any type of textual data. Here, $<:$ is the subtyping operator.

General-use plug-ins are those that accept more general types; if plug-in-specific types are used, other plug-in authors would have to co-develop against the plug-in to produce the requisite types, and the system would devolve into an extension-point based system.

C. Pipelines

The dataflow connection between two capabilities is a pipe. A pipeline is a combination of multiple capabilities using pipes. Pipes and pipelines have types, which is determined by combining the types of the capabilities. Note that one plug-in may expose multiple capabilities and appear multiple times in the same pipeline. In our example, `VersionControl` has the type

$$\forall L{<}: text.\ (source{<}L{>} \rightarrow source{<}L{>})$$

, and `PerformancePredict` has the type

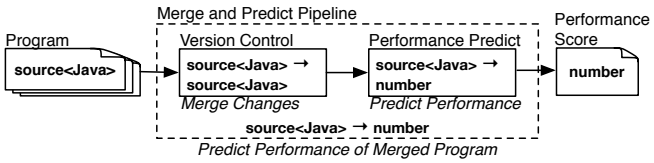

Figure 1. A pipeline that assesses the performance of a program after it is merged with the version checked into version control.

$$source{<}java{>} \rightarrow number$$

From `VersionControl`'s published capability, it is known that given a `java` source, it will produce `java` source. Since `PerformancePredict` consumes `java` source, we know the capabilities can be combined with a pipe. The resulting pipeline, shown in Figure 1, has the type $source{<}java{>} \rightarrow number$.

We propose to allow the end-user to design pipelines. We envision a graphical user interface à la LabVIEW (4) for creating a pipeline of capabilities. Given such an interface, the developer in our example could easily create the pipeline that provides post-merge performance analysis.

D. Filters

In our running example, our developer also wants to filter the refactoring suggestions from `RefactorSuggest` based on their performance. In general, a programmer may want to filter a collection of any type, such as action suggestions or program traces.

A filter removes elements from a set or list with element type τ using an *inclusion* predicate $\tau \rightarrow bool$. For each element in a collection (set or list), the element is included in the filtered collection if, and only if, the predicate is true for the element. Filters for sets have the type $(\tau \rightarrow bool) \rightarrow (\{\tau^*\} \rightarrow \{\tau^*\})$; filters for lists have the type $(\tau \rightarrow bool) \rightarrow ([\tau^*] \rightarrow [\tau^*])$. That is, they take an inclusion predicate, and produce a capability that filters collections using that predicate.

The key to defining a filter is defining the filter predicate. We envision that the user will be able to define a filter predicate just as they would a conditional breakpoint (which is a predicate over program states). More complex filter predicates may rely on the output of other plug-in capabilities or pipelines; a domain specific language might be introduced to handle these cases.

In our running example, the developer would like to use `RefactorSuggest` to generate refactoring suggestions (source changes). However, she would like to filter the actions so that the proposed set only includes actions that will not degrade performance. She can use the `VersionControl-PerformancePredict` pipeline created in the last section to create a filter. Note, however, that the output type of `RefactorSuggest` — $\{(source{<}java{>}, action)^*\}$ — is incompatible with the

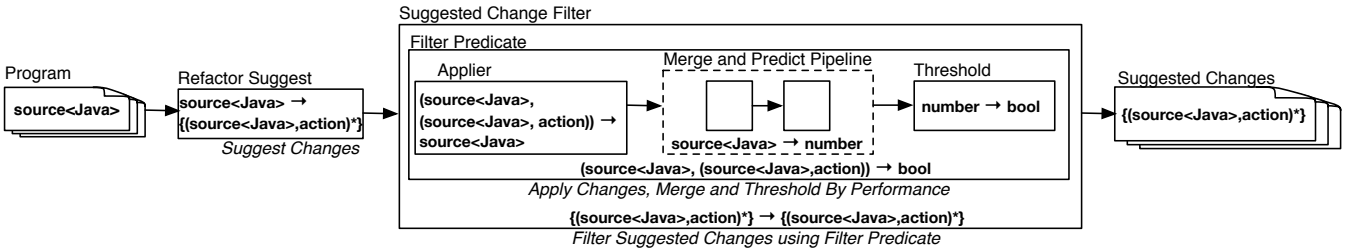

Figure 2. A pipeline that provides refactoring suggestions that do not harm the program's performance when merged. A set of raw refactoring suggestions are passed to a filter. The filter uses a predicate that applies a refactoring to produce a new source, and then analyzes the new source with a (reusable) pipeline that performs a merge and estimates the merged source's performance. When thresholded, the result is the set of refactoring suggestions that do not harm performance when merged.

input type $source<java>$ of the `VersionControl-PerformancePredict` pipeline.

The built-in applier capability: To solve this problem, we include a built-in *Applier* capability in the ecosystem, which applies a function to a value; the capability has type $(\tau_1, \tau_1 \rightarrow \tau_2) \rightarrow \tau_2$.

With the applier prepended to the merge performance pipeline, the user can use the pipeline to specify a predicate for `RefactorSuggest` based on performance values. For example, the predicate might be defined to block actions with corresponding performance values lower than some threshold. The end result, shown in Figure 2, is that changes suggested by `RefactorSuggest` are filtered according to the estimated performance of the program when the changes are merged with the repository.

III. Examples of Plug-In Synergies

This section describes synergies between existing plug-ins (and tools) that could be captured via the ecosystem.

A. Higher-Order Synchronization Feedback

Crystal is a speculative analysis tool that monitors a developer's local version control repository (5), providing a real-time alert when the local repository comes into conflict with the master repository *or another developer's local repository*. In addition to reporting textual conflicts, Crystal can report higher-order conflicts, such as when a merge would cause a compilation error, or the merged program would fail the test suite.

Our system enables the core functionality of Crystal to be built by the IDE's end-user (though a plug-in would have to provide capabilities for merging with other *local* repositories). This would make attaching additional analyses, such as a performance check, easy. The capabilities definitions for the pipelines that provide the higher level analyses are shown in Table II.

B. Dynamic (Testing) Feedback in Verification Mode

The Extended Static Checker for Java (ESC/Java2) plug-in (1) provides a "verification" perspective (mode) in Eclipse. In verification mode, the developer can invoke ESC/Java2 to find potential run-time errors in a program annotated with Java Modeling Language (JML) annotations. Warnings emitted by ESC/Java2 are marked in the source using Eclipse's standard marker mechanism. Essentially, the plug-in is a GUI wrapper for the ESC/Java2 tool itself; its capabilities are shown in Table II.

1) Contract Checking: If a developer has a correct program execution (e.g., from a passing test suite), that execution can be used to invalidate contracts. A tool providing this analysis might consume a program (and test suite), producing warnings for the invalidated contracts. The runtime checking capability is shown in Table II.

2) Contract Inference and Suggestions: Many tools exist which can infer JML contracts for a program. One such tool is Daikon (6), which uses a program trace to infer object invariants and method contracts via machine learning. Its capability is shown in Table II.

Since Daikon consumes a trace, it makes sense to use Daikon as part of a pipeline which includes a capability for producing traces. A good choice might be a plug-in for Chicory (Daikon's front-end), which has the interface $source<bytecode> \rightarrow source<trace>$. So, a pipeline consisting of: a compiler ($source<java> \rightarrow source<bytecode>$), Chicory, and Daikon would produce contract suggestions for a program.

C. Testing Feedback in Development Mode

For most developers, development and testing are two different activities. Saff and Ernst introduced continuous testing, a technique in which regression tests are run continuously during development with the aim of reducing wasted development effort (8). Their Eclipse plug-in (described in (9)) runs a project's JUnit test suite in the background whenever the developer saves the project; failed tests are reported alongside compilation errors.

In our model the end-user would be able to create the core continuous testing functionality simply by using the JUnit capability; the user could also easily create a filter to hide action suggestions that cause tests to fail.

D. Code Clone Feedback During Development

Code clones are syntactically or semantically similar program fragments that appear at multiple locations in a

Table II
PLUG-IN CAPABILITIES FROM SECTION III. FOR SIMPLICITY, WE ASSUME THAT TEST SUITES ARE PART OF THE PROJECT SOURCE.

Example Plug-In Capabilities	
Crystal (5) Merge Analysis Section III-A	
Merge Conflicts	$source<text> \rightarrow \{location<line>^*\}$
Compile	$source<java> \rightarrow \{label<text, string>^*\}$
Test	$source<java> \rightarrow \{label<test, bool>^*\}$
Esc/Java2 (1) Verification Section III-B	
Check	$source<java> \rightarrow \{label<text, string>^*\}$
Suggest	$source<java> \rightarrow \{label<text, string>^*\}$
Runtime Contract Checking Section III-B1	
Check	$source<java> \rightarrow \{label<text, string>^*\}$
Daikon (6) Invariant Detector Section III-B2	
Infer	$source<trace> \rightarrow \{label<method, string>^*\}$
Simian (7) Code Clone Detector Section III-D	
Detect	$\forall L <: text. (source<L> \rightarrow \{label<text, location < text >>^*\}$
Debugging Information Section III-E	
Code Churn	$source<text> \rightarrow \{label<line, number>^*\}$
Code Coverage	$source<java> \rightarrow \{label<line, number>^*\}$

codebase. Often introduced via copy and paste actions, code clones impair software maintainability since developers may fail to fix all the clones when fixing a bug (10). Detecting clones in real-time during development can point developers to existing code, preventing unnecessary duplication which can introduce new bugs (existing code has presumably been tested or used in production).

A bevy of tools and plug-ins exist to detect, visualize, and manage clones (11). Many detectors, such as Simian (7) can work with multiple file types, using special analyses when possible. The published capability can be written generically (see Table II), though the plug-in performs a type-specific analysis.

E. Augmenting Debugging Mode

The debugging mode typically contains views of the program source, stack trace, and expression evaluation. In the plug-in ecosystem, users can easily augment these views with information about code churn or test coverage.

Empirical evidence suggests that areas with relatively high code churn are likely to contain more bugs (12). Similarly, areas with low or no test coverage may be more likely to contain bugs (13). A test suite cannot detect a bug if it does not execute the line that contains the bug.

A developer can leverage these insights when debugging by overlaying the churn or test coverage in the source view, e.g., via a heat map, in order to focus her efforts on locations with high churn and/or low coverage. The capabilities, which produce numeric scores for lines, are shown in Table II.

IV. DISCUSSION

In this section, we address the implications and limitations of the model. By and large, the limitations can be addressed by tracking additional information in the model.

A. Informing Plug-In and Tool Design

In the introduction, we described a conceptual model of IDEs in which the IDE and plug-ins consist of modes, analyses, views, and actions. Table I outlined where existing features, and the capabilities described in Section III fall. A benefit of presenting the information in this form is that it naturally poses questions of the form "What would happen if item X in box A was also in box B?" Asking this question for code churn led us to suggest marking code churn in the source view when debugging in Section III.

In Section II we presented a type system for plug-in interaction. The type system eliminates nonsensical plug-in combinations. Additionally, the type system provides a principled way to explore the plug-in (tool) design space, allowing users to find useful, but non-obvious combinations of plug-ins (tools) with compatible capabilities. When combined with our IDE model, the result is a powerful tool for inventing new program analyses and tools.

B. Change Analysis

In the model presented, a program analysis produces a result τ from a single $source$. In many cases, the user would like a comparative analysis instead. A simple way to perform comparative analysis would be to run the analysis on both versions of the source, comparing the results of the analysis; we could provide an interface for the end-user to create custom change analyses, just as they can create custom filters.

More expressive change analyses would rely on information about the specific action; this would involve adding metadata to the $source \rightarrow source$ type, which is currently just a function type (arrow).

C. Performance

Both the IDE and developer have bounded resources. Therefore, some capabilities are neither tractable nor desirable — a developer likely will not wait for the entire test suite to run when viewing a refactoring preview.

Minimal Analyses: In many cases, a partial analysis that targets the user's active focus (e.g., the active source file) is sufficient and sound. In these cases, a plug-in should publish both full and partial analyses. Code clone detection is a prime example. While detection is typically performed over an entire project, clone search using the active file as a query is likely to be sufficient during development. Unfortunately, providing the partial contribution can require additional analyses, e.g., determining which tests must be rerun by analyzing a callgraph.

Planning and Reuse: Like a database query processing engine, the IDE can and should plan pipeline evaluation to minimize resource usage. Since capabilities are assumed to be pure in the model presented, opportunities for result reuse should abound. Relaxing the purity assumption would severely hamper planning unless the impurities are modeled properly (e.g., with a form of separation logic).

Developer Control: Users understand their information needs best. When enabling a plug-in's capability, the user should be able to specify at least two things:

- Grace Time: the maximum time to wait for the capability, and
- Background: whether or not to proceed until the capability is complete.

Grace time is useful for pessimistic analyses; background mode is useful for pipelines that contribute to a view that can display results asynchronously.

V. Related Work

The core idea presented in this paper is not new; Unix-based operating systems allow users to compose the inputs and outputs of programs in a data-driven manner via piping. Windows PowerShell offers similar with an object-pipeline built on top of .NET objects. Our formalization reconciles the pipepline concept with plugins and the IDE.

Speculation: Speculation is an area of research which aims to inform developer decision by anticipating developer actions and then performing analysis on the concrete artifacts generated by speculatively applying each action (14). Both our motivating example and Crystal (5), described in Section III-A, are examples of speculative analyses. The ecosystem described in this paper is complementary to speculation; we've shown that the system enables the end-user to define their own speculative analyses.

Mashups: Mashups are (typically) websites that combine information from multiple sources to produce new insights. Yahoo Pipes is a visual programming language designed to enable lay-users to create mashups by combining existing web data feeds (15). Domain-specific visual programming languages are common in dataflow programming (16). The primary difference between mashups and our design is that mashup data sources are typically independent.

Formal Plug-In Architecture Models: Chatley et al. formalize a general plug-in architecture in Alloy for self-assembling systems (17). The architecture, which is based on typed "pegs" and "holes" is designed to enable a system to automatically combine plug-ins at runtime based on their interfaces. While this basic concept applies to our work as well, their formalization depends heavily on rules for self-assembly. Our model benefits from having the developer as a final oracle of whether or not a combination is sensible.

VI. Conclusion

We have presented a plug-in ecosystem for IDEs that enables the *end-user* to combine data-driven plug-in capabilities to create new insight. Each plug-in publishes its capabilities to the IDE. Typing rules prevent nonsensical plug-in combinations, and inform the design of new tools and plug-ins providing capabilities that other plug-ins can *independently* leverage.

Acknowledgments

We thank Colin Gordon for his comments on an early draft of this paper. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-0718124.

References

[1] (2012, Feb) ESC/Java2. [Online]. Available: http://kindsoftware.com/products/opensource/ESCJava2/

[2] B. C. Pierce, *Types and programming languages*. Cambridge, MA, USA: MIT Press, 2002.

[3] (2012, Mar) Quick Fix. [Online]. Available: http://bit.ly/eclipse-quickfix

[4] (2012, Mar) NI LabVIEW. [Online]. Available: http://www.ni.com/labview/

[5] Y. Brun *et al.*, "Proactive detection of collaboration conflicts," in *Proc. ESEC/FSE'11*, 2011.

[6] M. D. Ernst *et al.*, "The Daikon system for dynamic detection of likely invariants," *Sci. Comput. Program.*, vol. 69, no. 1–3, pp. 35–45, Dec. 2007.

[7] (2012, Mar) Simian - Similarity Analyser. [Online]. Available: http://www.harukizaemon.com/simian/

[8] D. Saff and M. D. Ernst, "Reducing wasted development time via continuous testing," in *Proc. ISSRE'03*, 2003.

[9] ——, "Continuous testing in Eclipse," in *Proc. eTX'04*, March 2004.

[10] E. Juergens *et al.*, "Do code clones matter?" in *Proc. ICSE'09*, 2009.

[11] C. K. Roy *et al.*, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. Comput. Program.*, vol. 74, pp. 470–495, May 2009.

[12] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proc ICSE'05*, 2005.

[13] W. E. Wong *et al.*, "Effective fault localization using code coverage," in *Proc. COMPSAC'07*, 2007.

[14] Y. Brun *et al.*, "Speculative analysis: Exploring future states of software," in *Proc. FoSER'10*, 2010.

[15] (2012, Mar) Yahoo Pipes. [Online]. Available: http://pipes.yahoo.com/pipes/

[16] W. M. Johnston *et al.*, "Advances in dataflow programming languages," *ACM Comput. Surv.*, vol. 36, pp. 1–34, March 2004.

[17] R. Chatley *et al.*, "Modelling a framework for plugins," in *Proc. SAVCBS'03*, 2003.