# Speculative Reprogramming

Marc Palyart
UBC, Canada
mpalyart@cs.ubc.ca

Gail C. Murphy
UBC, Canada
murphy@cs.ubc.ca

Emerson Murphy-Hill
NCSU, USA
emerson@csc.ncsu.edu

Xavier Blanc
U. of Bordeaux, LaBRI, France
xblanc@labri.fr

## ABSTRACT

Although software development involves making numerous decisions amongst alternatives, the design and implementation choices made typically become invisible; what a developer sees in the project's artifacts are the end result of all of the decisions. What if, instead, all of the choices made were tracked and it was easy for a developer to revisit a point where a decision was made and choose another alternative? What if the development environment could detect and suggest alternative choices? What if it was easy and low-cost to try another path? We explore the idea of speculative reprogramming that could support a what-if environment for the programming stages of software development.

## Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Integrated environments

## Keywords

Design alternatives, What-if

## 1. INTRODUCTION

Software programming today is largely a flat-line activity. Although a software developer implementing a design makes many choices, such as which library to use, which data structures to use and so on, these choices are seldom captured; the code committed to the repository is typically the final end choice.

To support programming as the tree-activity it is, we propose speculative reprogramming. In this approach, design and implementation alternatives that arise are explicitly represented and a developer is enabled to easily investigate multiple alternatives for the program. The alternatives that arise may be specified explicitly by the developer, may be captured implicitly as the developer works or may be determined implicitly based on analysis of the program and available resources (e.g., the web). In speculative reprogramming, tool support makes it easy to explore a different alternative path by allowing back-tracking to an alternative point and rolling forward of the program as automatically as possible. Tool support also enables automatic analysis of the effect of an alternative decision, say by determining that the performance of the program would be improved with a different alternative. The intent is to make it easier for developers to explore the design space of their programs by:

1. automatically recording alternatives that the developer explicitly explores,
2. automatically discovering and recording alternatives that the developer did not explicitly explore,
3. making it easier to roll back prior choices and replace them with alternatives, then rolling forward subsequent changes, and
4. estimating the impact of past design choices and alternatives on the current system.

To make the overall approach more concrete, we begin with an example of how speculative reprogramming may aid a developer (Section 2). We then describe key aspects of the speculative reprogramming design space (Section 3). Making the approach in one or more forms a reality involves many research challenges (Section 4) but can also build on existing work (Section 5). We summarize with a first suggested approach (Section 6).

Publication Rights Licensed to ACM. This is the authors' version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version will be published in the 22nd ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-22) Proceedings.

## 2. AN EXAMPLE

Consider a software developer implementing an application in Python. As the developer is implementing the central algorithm of the application, the developer realizes that the implementation could benefit from parallelism. The software developer is relatively new to using the Python language so searches the web for possible libraries to support multi-threading. From a web search, the developer finds information on StackOverflow[1] that raise various possibilities, including threading support built into Python, a multiprocessing API, and parallel Python.

With today's tools, the developer makes a decision of which parallelism approach to use, say built-in support, and development continues. However, the point at which alternatives were available and where a decision on parallelism was made is typically lost. If a developer later finds that a different choice may be preferable, the developer must manually re-determine the other alternatives, must manually determine the code impacted by the change and must manaully revise the code to use the newly chosen alternative.

[1]e.g., `http://stackoverflow.com/questions/2987980/parallelism-in-python`

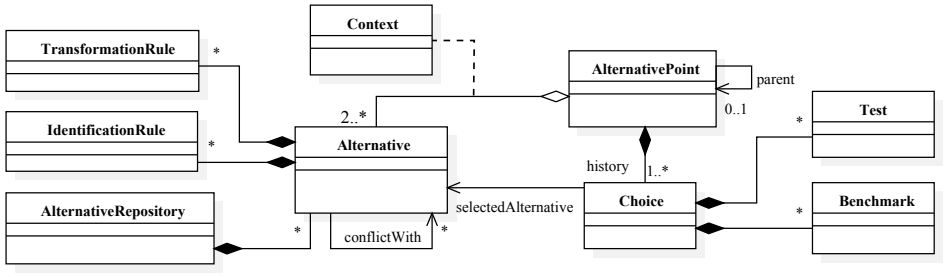


**Figure 1: Class diagram showing the main concepts of the approach**

Contrast this manual effort with what happens in a speculative reprogramming environment. In such an environment, the tooling detects, based on the contents of the web search performed by the developer to investigate parallelism, that alternative solutions are available. The tooling determines the possible alternatives and suggests the alternatives to the developer as shown in Figure 2.

**Alternative 1: Python threading support**

| Non-automated | See |
|---|---|
| • import the threading module<br>• modify classes to inherit from Thread<br>• ... | • http://inventwithpython.com/blog/2013/04/22/multithreaded-python-tutorial-with-threadworms/ |

**Alternative 2: Multi-processing API**

| Non-automated | See |
|---|---|
| • drop-in for Python threading but import multiprocessing | • http://pymotw.com/2/multiprocessing/basics.html#importable-target-functions |

**Figure 2: Alternative Suggestions**

The developer views the two alternatives and decides to start with the first. Although the developer reads in the suggested web pages that there may be effects of a global interpreter lock, the approach must be applied before the second alternative can be considered. The developer proceeds with the implementation. During this implementation work, the developer makes a number of other choices, such as data structures. All of these decisions are recorded by the speculative reprogramming environment.

Much later in development, the developer determines that the performance of the application is not acceptable. The developer writes tests that express the performance constraint and requests the speculative reprogramming environment to automatically assess past choices to see if another alternative might help meet the constraint. The environment looks for points in the decision tree with code related to the code executed by the performance tests. As it backtracks the tree, it finds the decision made to pursue the threading approach. The tooling again evaluates for alternatives at the associate code locations and finds transformation rules that can automatically transform the program from using the threading alternative to using the multiprocessing library. The tooling applies the transformations, is able to re-execute the performance tests and demonstrate that the application now meets the stated performance constraints. Meeting the performance required little of the developer. At low-cost, alternate development paths from a previous decision could be investigated and an appropriate path found.

With this example, we have sketched the speculative reprogramming approach, which may be realized in a number of ways. At the most manual point in the design space, tooling could be provided to allow a developer to explicitly mark alternative choices, to view the design tree and to return to previous points in the design tree with conservative transformation rules to roll a program forward through different alternatives through only semantics preserving paths. In a highly automated point in the design space, tooling could be provided that relies on developer specified test suites to automatically choose and form different programs that meet the test suite constraints. Many other design points are also possible.

# 3. APPROACH

Key elements of the speculative reprogramming approach include *alternative points* and *alternatives*. Figure 1 shows how these key elements relate.

An *alternative point* represents a specific state of the source code in which multiple alternate ways forward exist.

An *alternative* is a design solution (e.g., a data structure choice or a framework choice). Several alternative points in an application may refer to the same alternatives (e.g., the `TreeMap` and `HashMap` alternatives). An alternative is specialized to a particular case through a *context*, which may add such information as location information. The definition of an alternative includes *identification* and *transformation* rules. An identification rule enables the detection of where the alternative might apply within a provided set of source code (e.g., find all occurrences of a `HashMap` in a Java program). A transformation rules enables the automatic application of the alternative to a provided set of source code (e.g., use `TreeMap` as implementation for the `Map` interface) or describes steps a developer can take to apply the alternative manually (e.g., see Figure 2). To go from one alternative to another, identification rules from the current alternative may be used with transformation rules from the target alternative.

The alternatives that may be applied to a given set of source code are stored in *alternative repositories*. These repositories may be private or may be created by mining resources on the web, such as open source code repositories, search engine, and Q&A websites.

For any given program on which speculative reprogramming is used, the history of *choices* of alternatives for a specific alternative point are stored. All the data that can document and justify the choice is also stored. For example this can range from metrics, recommendations, source code

to data generated by the tooling (*test*s or *benchmark*s run to assess the set of alternatives).

# 4. CHALLENGES

Making speculative reprogramming a reality for developers requires research in several areas.

## 4.1 Defining Alternative Points

Alternative points may take many forms; here are a few examples:

- what data type should be chosen for a variable? (e.g., should this variable be declared as a float or a double based on the application's constraints?)
- what implementation should be chosen for an interface? (e.g., is a `HashMap` or a `TreeMap` suitable for implementing the Java `Map` interface in this application?)
- which API should be used to provide a desired commodity functionality? (e.g., should the application use *org.json*, the original library, or a newer on such as *Jackson*?)
- which algorithm should I use? (e.g., should I use a search algorithm tailored for my specific data structure or a more general search algorithm?)
- which architectural design is better suited for my application? (e.g., should I use a heartbeat or a ping architecture ?)

Existing literature may be mined for kinds of alternative points. As just one example, Parnas' seminal paper on modularity discusses a number of alternatives that arise in even a simple design exercise [8]. More analytic and empirical research is needed to determine a useful list of the kinds of alternative points and the ways in which the points are related. Which alternative points are a superset of others? What can be learned from observing software developers at work with regards to alternative points?

## 4.2 Detecting Alternative Points

One form of speculative reprogramming could be based on having a developer explicitly indicate when an alternative point is reached. This approach would require a developer to self-reflect during programming, a difficult activity. Speculative reprogramming will likely be more usable if alternative points are auto-detected, perhaps by:

1. monitoring a developer's activity in the editor and inferring decision points based on the kinds of edits, such as changing the type of a variable,
2. monitoring a developer's activity across a range of applications and using natural language processing to identify concepts that suggest alternatives are being considered (e.g., as in Section 2 when a web search related to concepts being programmed occurs), or
3. matching program edits to transformation rules previously mined from other programs.

Work on live programming that is emerging may be a good source of alternative detection techniques. For example, Soares et al.'s work on determining if edits to a program constitute a semantics-preserving behavioural change or not could help determine whether a change indicates a refactoring or a functional change to the code [9]. Another direction may be to build on feature location techniques to allow investigation of similar features [4]. However, more research is needed to formalize and implement the detection of alternative points.

## 4.3 Defining and Applying Alternatives

Each alternative has associated identification and transformation rules. Ideally, alternatives would be defined automatically by mining existing programs. By comparing two versions of a program where a specific activity was undertaken, such as changing from one library to another, it may be possible to derive alternatives (e.g., [10]). Alternatives might also be seeded through analysis of question and answer sites, such as *StackOverflow*, and completed manually by developers. Techniques would need to be developed to represent the likely completeness of a given alternative (e.g., is the application of the alternative semantics preserving) and to communicate the alternative and its completeness to the developer.

Various technologies may help to apply an alternative. For example, if a developer selects to change from the use of one framework to another at an identified alternative point, the approach by Dagenais and Robillard to recommend adaptive changes that enable a software system to adapt to a new technological environment may be helpful [3].

Even more challenging is the desire in speculative reprogramming to re-apply a different alternative to a previous point in the decision history of the program. There are two basic choices: 1) roll the program back, apply a different alternative and roll other changes after the alternative point forward, and 2) apply the alternative to the existing version of the program. Both approaches are fraught with challenges in trying to automatically determine how the affected code may impact other choices made in the implementation. Existing research on how programmers backtrack (e.g., [11]) may provide a basis on which to start this research direction. It will also be challenging to determine how to interact with the developer in altering the program to different points in time where the semantics may or may not affected by different choices.

## 4.4 Assessing Alternatives

Alternatives may be assessed either manually by a developer, perhaps by viewing the identification and transformation rules associated with an alternative, or automatically by evaluating the result of applying an alternative against an objective measure. The objective measure may take several forms. As in Section 2, the objective measure may be a test suite specified by the developer for the application. As another example, the developer may annotate the program with constraints, such as performance ranges or invariants that are to be maintained.

A particular challenge when assessing alternatives is whether they are considered independently (i.e., select the best alternative without consider other choices made later in development) or are interactions between alternatives considered? The latter may not be possible depending on search space of possible alternative combinations.

## 4.5 Storing Decisions and Alternatives

An alternative point references alternatives and is specific to a revision of the source code. Ideally, choices made between alternatives would be stored. The tree of decision points and choices provides a new kind of software development documentation, for instance, potentially recording that a developer chose to use one library over another.

Storing the decision points and alternatives across a development team holds several challenges. Should some decision

points and alternatives be stored locally for a given developer? How long are decision points relevant? Should the decision tree be erased after a day, a month, a year? How should decision trees be shared amongst a team? Should a developer be able to flush irrelevant decision points? There is little work that we know of that provides a basis for this research direction.

## 5. RELATED WORK

Speculative reprogramming might be viewed as a case of search-based software engineering [6, 5] in which the basic idea is to automatically generate options for solving a software engineering problem with the search through options directed by a fitness function. The proposed approach differs from search-based engineering by focusing equally on helping a developer determine, express and search a design space and on moving to different points in the design space. Unlike search-based engineering, the intent is not solely to reach a desired optimal point in the search space, but to allow the developer to fluidly move between possible points in the search space. For example, a search-based software engineering approach to refactoring (e.g., [7]) can generate alternatives of refactorings to apply and use a fitness function, such as a weighted measure of various coupling and cohesion metrics, to decide on which refactoring to apply. This approach, like others in search-based software engineering, drives towards a goal, a refactored program with good quality characteristics as opposed to enabling back-tracking and exploration to various points in the search space.

Cadar et. al discuss the new possibilities enabled by the availability of a great range of computational resources, such as the ability to run multiple versions of an application in parallel, allowing automatic optimization of parameters to meet non-functional reuqirements [2], such as performance. Similar to Cadar et. al, we consider alternatives; our work differs in focusing on the how to create and enable a developer to interact with an explicit program design tree.

The idea of speculative reprogramming was inspired, in part, by Brun et. al's work on speculative analysis, in which the potential consequences of future actions of a developer are evaluated and provided to a developer [1]. Speculative reprogramming is not limited to the future development states but focuses on capturing the tree of potential development paths and enabling a developer to analyze and revert alternative development paths based on past decisions. The two approaches are similar in aiming to ease a developer's investigation of alternate development paths.

## 6. THE WAY FORWARD

Achieving a highly-automated speculative reprogramming environment for a different kinds of alternative points and for different kinds of programs requires solving many technically hard problems. Achieving exploratory speculative reprogramming environments for a small set of alternative points for smallish programs is within reach. We propose that work on speculative reprogramming begin by building a speculative reprogramming environment aimed at novice Java developers that has automated detection of alternative points and that supports two kinds of alternatives: one with automated transformation rules (e.g., switching between implementations of data structures such as `Map` from the Java libraries) and one with manual transformation rules (e.g., moving from one JSON library to another). The limited number of alternatives renders the automatic detection of alternative points possible. Such an environment could be used to conduct laboratory experiments to understand what support a developer needs to fluidly move through the tree-based alternative development paths possible and would make it possible to assess the utility of the idea.

Working in a speculative reprogramming environment would make it possible for a software developer to ask "what-if" more often and to lower the cost of each of those questions.

## Acknowledgements

We thank the 2014 Bellairs Pro-K workshop organizers for providing a venue that sparked this paper. We thank Jonathan Sillto for his contributions to the formative ideas. This work was funded in part by NSERC and in part by the National Science Foundation under Grant No. 1217700

## 7. REFERENCES

[1] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Speculative analysis: exploring future development states of software. In *Proc. of the FSE/SDP Work. on Future of Soft. Eng. Research*, pages 59–64. ACM, 2010.

[2] C. Cadar, P. Pietzuch, and A. L. Wolf. Multiplicity computing: a vision of software engineering for next-generation computing platform applications. In *Proc. of the FSE/SDP Work. on Future of Soft. Eng. Research*, pages 81–86. ACM, 2010.

[3] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. In *Proc. of the 30th Int'l Conf. on Soft. Eng.*, ICSE '08, pages 481–490, New York, NY, USA, 2008. ACM.

[4] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.

[5] M. Harman. The current state and future of search based softnnware engineering. In *2007 Future of Soft. Eng.*, FOSE '07, pages 342–357. IEEE Computer Society, 2007.

[6] M. Harman and B. F. Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833–839, 2001.

[7] M. O'Keeffe and M. O'Cinneide. Search-based software maintenance. In *Proc. of the Conf. on Soft. Maint. and Reengineering*, pages 249–260. IEEE Computer Society, 2006.

[8] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[9] G. Soares, E. Murphy-Hill, and R. Gheyi. Live feedback on behavioral changes. In *1st Int'l Workshop on Live Programming*, pages 23–26. IEEE, 2013.

[10] C. Teyton, J.-R. Falleri, and X. Blanc. Automatic discovery of function mappings between similar libraries. In *20th Working Conf. on Reverse Engineering*, pages 192–201, 2013.

[11] Y. Yoon and B. A. Myers. A longitudinal study of programmers' backtracking. In *IEEE Symp. on Visual Lang. and Human-Centric Comp.*, 2014.