Heuristic Evaluation of Programming Language Features: Two Parallel Programming Case Studies

Caitlin Sadowski Sri Kurniawan
University of California at Santa Cruz
{supertri,srikur}@cs.ucsc.edu

Abstract

Usability is an important feature for programming languages, and user evaluations can provide invaluable feedback on language design. However, user studies which compare programming languages or systems are often very expensive and typically inconclusive. In this paper, we posit that discount usability methods can be successfully applied to programming languages concepts such as language features. We give examples of useful feedback received from applying heuristic evaluation to two language features targeted at parallel programming.

Categories and Subject Descriptors H.1.2 [*Models and Principles*]: User/Machine Systems; D.1.3 [*Programming Techniques*]: Concurrent Programming

General Terms Human Factors

Keywords Programming Languages, Usability, Heuristic Evaluation

1. Introduction

Parallel and concurrent programming is extremely difficult [1, 13]. Parallel and concurrent programming is also increasingly pervasive; concurrency is a key component to all reactive applications, and the recent prevalence of multicore hardware has made exploiting parallelism a major aspect of performance optimizations [1]. A plethora of language features aid parallel and concurrent programming; since parallel and concurrent programming is so hard, evaluating and improving the usefulness of these language features could have a big impact.

As an exercise, one of the authors of this paper modified Nielsen's 10 heuristics [15], plus the 13 tradeoffs which make up the cognitive dimensions framework [9], to create a selection of heuristics for evaluating language features, and then used those modified heuristics to evaluate the language feature of machine-checkable `yield` annotations [22], described below. The issues uncovered by performing this simple usability engineering method were striking; thinking about `yield` annotations in terms of heuristics uncovered several research questions related to making `yield` annotations a usable feature. By identifying usability issues early on, it may be possible to improve language features before they are widely deployed. We recruited four additional evaluators and examined one additional language feature to test the hypothesis that heuristic evaluation could be a useful tool for programming languages researchers.

1.1 Language Features

In this paper, we investigate using discount usability methods, specifically heuristic evaluation, to identify potential problems in language features which have been proposed to help programmers reason about multithreaded code. We focus on two proposed language features: `atomic` annotations [7] and `yield` annotations [22].

The `yield` annotations represent the points at which context switching can affect the results of executing code. For example, take a look at the following code snippet:

```
public class Example {
  int x;
  public void foo() {
    float tmp1 = x;
    /* yield; */
    float tmp2 = x;
  }
  public void bar() {
    x = 3;
  }
}
```

The "`/* yield; */`" annotation represents the fact that another thread could change the value of `x` at this program point, for example, by executing the `bar()` method. Note that these annotations do not change the code behaviour, they just document the points of potential thread interference.

PLATEAU'11, October 24, 2011, Portland, Oregon, USA.

In contrast, `atomic` annotations represent blocks of code which do not contain thread interference internally. Here is the example from above rewritten with `atomic` annotations:

```
public class Example {
  int x;
  public void foo() {
    atomic{ float tmp1 = x; }
    atomic{ float tmp2 = x; }
  }
  atomic public void bar() {
    x = 3;
  }
}
```

Roughly speaking, `yield` annotations would correspond to the points where atomic blocks meet, in code that is entirely covered by atomic blocks. As before, `atomic` annotations do not change the code behaviour.

For both types of annotations, evaluators were informed that programmers could use these annotations in two ways. First of all, programmers could annotate methods with `yield` or `atomic` and then analyze their annotated programs to check if the annotations were correct. Several such checkers exist in the research literature for `atomic` annotations (e.g. [7]); a `yield` annotation checker has also been published [23]. As an alternative use, a program (or another programmer) could annotate a program with correct annotations, perhaps via a `yield` annotation inference tool [23]. Future programmers could use these annotations to reason about the code.

2. Related Work

Usability testing has sometimes been used to inform the design of programming languages. The most notable example of this is the HANDS language for children [19]. This language involved HCI principles as an integral part of the language development process; after the language was developed, the authors performed a user study to evaluate the addition of some key language features by comparing language versions with and without the features.

Discount usability [16, 15] is a set of fast techniques for evaluating the usability of a system. These techniques include scenarios, card sorting, and heuristic evaluation. Discount usability engineering has been used in a variety of contexts, including agile development [12]. However, discount usability methods have also been criticized, particularly for not having good coverage metrics or a clear way to analyze the results for false positives or false negatives [8].

In this paper, we use a variant of heuristic evaluation [17]. Heuristic evaluation has been successfully used in a variety of contexts, ranging from evaluating mission-critical software with a large team [3] to evaluating games [20], and has been shown to be effective at finding severe usability problems [10]. Nielsen's heuristics have also been used to categorize prior research focused on usability and novice programming systems [18].

We base some of our heuristics on the cognitive dimensions framework [5]. This framework was originally presented as a set of conceptual categories which highlight design tradeoffs in visual programming languages [9]. These dimensions were later formulated as a questionnaire for users [2]. Other researchers have used cognitive dimensions to evaluate programming languages [4], or formatively in developing a language feature [11].

2.1 Evaluating Heuristics

Some prior research has focused on evaluating a set of proposed domain specific heuristics: a full evaluation of the heuristics presented in this paper remains an area for future work.

Desurvire et al. [6] focused on evaluating the playability of games. They evaluated the set of heuristics they developed by comparing usability problems identified through their heuristics with the usability problems identified through a user study.

Mankoff et al. [14] developed a set of heuristics for ambient displays by starting with Nielsen's heuristics and modifying them for the domain. In an initial study, participants rated the identified heuristics in terms of relevance to ambient displays and suggested additional heuristics. After this refinement stage, the set of heuristics was used to evaluate two displays possessing a list of known usability problems.

Sim et al. [21] developed a set of heuristics for analyzing computer assisted assessment (CAA) devices by refining a set of problems with CAA, gleaned from a large survey and two heuristic evaluations, into categories. They also established that a heuristic evaluation performed using Nielsen's original set of heuristics did not identify enough of these problem categories.

3. Methodology

We start with Nielsen's 10 heuristics [15], plus the 13 tradeoffs which make up the cognitive dimensions framework [9], rewritten as heuristics. Two evaluators went through this list of 23 potential heuristics to see whether they could apply them to yield annotations. We updated, merged, or deleted heuristics which did not make sense in the context of a language feature. The final list of heuristics is displayed in Table 1; the first 7 started with the cognitive dimensions framework and the last four started with Nielsen's original list of heuristics.

Using these 11 heuristics we ran a heuristic evaluation with five evaluators, including one of the authors of this paper. The evaluators were computer science graduate students with industry experience and were all experienced with parallel and concurrent programming. All evaluators received a short lecture on heuristic evaluation before beginning, and recorded their impressions on a worksheet. Although heuristic evaluation is usually performed by usability experts, we

H1	**Abstraction Gradient**	Does this feature adequately cover the gradient between minimum and maximum levels of abstraction in a program? (e.g. the gradient from low-level code details to interface-level details)
H2	**Consistency**	Does this feature have consistent meaning?
H3	**Error-Proneness**	Does the notation for the language feature induce "care-less mistakes"? Is there a match between notation in the system and how similar language is used in the real world?
H4	**Hidden Dependencies**	Are all dependencies clear with this feature? Could local changes have confusing global effects?
H5	**Premature Commitment**	By using this language feature, do programmers have to make decisions before they have the information they need?
H6	**Progressive Evaluation**	Can a partially-complete application of this feature provide feedback on "How am I doing"?
H7	**Viscosity**	How much effort is required to perform a single change involving this feature?
H8	**Flexibility & Efficiency**	Is this feature effective across the gradient between novice and expert programmers?
H9	**Aesthetic Design**	Is feature notation simple and concise?
H10	**Error Recovery**	If there is an error in language feature usage, are precise, constructive error messages in plain language presented?
H11	**Documentation**	Is documentation describing the feature concise, concrete, and relevant?

Table 1. Eleven Language Feature Heuristics

wanted to explore whether language researchers could find new problems with this exercise.

We first had evaluators evaluate `yield` annotations with the heuristics, and then evaluate `atomic` annotations; evaluating each annotation took 15-20 minutes. All evaluators were familiar with research on `yield` and `atomic` annotations before the study. Since they had previous experience with these annotations, we did not need to provide examples of their usage during the evaluation.

Taking inspiration from a study about developing a heuristic evaluation for video games [20], we asked the four non-author evaluators to describe strengths and limitations of using the supplied heuristics to evaluate language features. We also asked evaluators to identify any heuristics they found particularly useful or particularly confusing. Lastly, we asked evaluators if using the heuristics led to finding problems they would otherwise have missed, and whether the heuristics gave them new perspectives on the language feature.

4. Results

Evaluators identified 5 problems for both `atomic` annotations and `yield` annotations; of these problems, we are not aware of prior discussion of two of them in the research literature. Evaluators identified 7 additional problems when focusing on `yield` annotations, 5 of which are (to our knowledge) new problems. Evaluators also identified 5 additional problems when focusing on `atomic` annotations, two of which are (to our knowledge) new problems.

Nielsen's original heuristic evaluation entails that evaluators impose a severity rating for problems along a four point scale between cosmetic and catastrophic. In this study, evaluators rated the same problems at various places on Nielsen's scale, and did not find the scale to be helpful. We have omitted the severity rating assigned from the following discussion since ratings were very mixed and we feel that the identification of new problems is the major research contribution of this work. In fact, severity ratings are usually omitted from domain specific heuristics due to their variability [21]. For each heuristic, at least one evaluator identified a problem using that heuristic. Because of this, we believe that the list of heuristics could be expanded, but should not be compressed.

In the following discussion, we append the issues identified with an ID for each participant/heuristic pair that mentioned a particular issue. For example, a problem with (P2: H2) after it was identified by the second participant using the "Consistency" heuristic.

4.1 Problems identified with both atomic and yield annotations

1. Evaluators were concerned about other definitions of the words "atomic" and "yield" leading to confusion. (P2: H2, P3: H3, P4: H3)

   "Yield is an overloaded word; not self-documenting."

   We feel that the choice of terminology can have a big impact in understanding concepts; we recommend that possible sources of notational confusion should be discussed more in the literature.

2. Evaluators were concerned about the usefulness of tool feedback for missing `yield` or `atomic` annotations. (All evaluators: H10) We feel that usability of tool feedback is underdiscussed in the literature. In particular, evaluators were worried that feedback would not adequate explain *why* an error occurs.

3. Evaluators realized that missing `yield` annotations or inaccurate `atomic` sections may lead to strange results

and may not represent real bugs. (P2: H6, P3: H4/H7, P5: H6)

> *"It is unclear how bugs related to inadequate yield annotations will actually manifest."*

> *"Knowing something is non-atomic may result in making changes to an entirely separate part of the program."*

This problem represents an area of active research.

The remaining two problems are not, to our knowledge, discussed in the research literature.

1. Evaluators were concerned about how `atomic` or `yield` annotations relate to evolving code. (P2: H5/H7, P5: H5) Although `yield` annotations may be useful in program evolution [22], this aspect needs to be explored further. We are not aware of research which looks at how atomic blocks of code evolve over time.
2. All evaluators were concerned about the lack of documentation for these features, beyond research papers. (All evaluators: H11) By failing to convey the intent and meaning of a language feature, lack of documentation may adversely affect the usability of that feature and hinder its use.

4.2 Problems identified with yield annotations

Two problems identified with `yield` annotations represent general issues in annotation checking tools and so are not unique to this particular language feature.

1. Evaluators were concerned about the impact of false positives or false negatives in the checker on the usefulness of `yield` annotations. (P1: H3, P5: H3)
2. An evaluator was worried about users ignoring `yield` annotations. (P1: H3)

The remaining five problems are not, to our knowledge, discussed in the research literature.

1. Evaluators worried about a false sense of security caused by correct yield annotations. (P1: H3, P5: H5)
2. Evaluators identified a problem in the difference between local and global reasoning with yields. (P1: H4, P2: H1/H4, P3: H1, P5: H3)

> *"Yields are specific to a line number; but how do you reason about methods that may contain yields inside without looking at the code?"*

3. Evaluators were worried that too many yields could clutter code, be confusing, or be annoying to write. (P1: H6/H7, P3: H8)
4. One evaluator pointed out that: (P3: H6)

> *"Partially correct set of yield annotations is not worth very much."*

5. Evaluators thought the word yield might be too minimal, and that perhaps additional information could be included in the notation. (P1: H10, P2: H2, P5: H3)

> *"Which variables are affected by the yield points?"*

4.3 Problems identified with atomic annotations

Three identified problems with atomic annotations are also discussed in the literature on yield annotations [22].

1. Atomic sections create bimodal reasoning. (P3: H1, P4: H1)

> *"Atomic forces programmers to think about code as if [it is either] in atomic or out of atomic."*

2. Evaluators pointed out that `atomic` annotations based on syntactic blocks are limited. (P3: H2, P4: H9)
3. Evaluators were concerned about the global impact of atomic or non-atomic sections. (P3: H4/7, P4: H7, P5: H1)

> *"How does making one method atomic or non-atomic change the entire program behaviour?"*

The remaining two problems are not, to our knowledge, discussed in the research literature.

1. Evaluators wondered how `atomic` annotations could be applied at the class level. (P1: H1, P3: H3)

> *"Does an atomic interface mean a safe interface?"*

2. Evaluators noted that `atomic` annotations were only useful if correctly applied. (P1: H3, P3: H8, P4: H8, P5: H8)

> *"[Atomic annotations are] easy to add, hard to add well."*

5. Discussion

This initial evaluation, though small, uncovered several new research issues in a short amount of time. However, the small number of evaluators and the ambiguity of interpretation of some of the heuristics represent threats to validity of this study; further studies are needed to refine and build confidence with the set of language feature heuristics presented here.

Multiple evaluators found that the heuristics which focused on error-proneness or progressive evaluation were particularly useful; we think these are important heuristics for programming language researchers to consider when developing language features or tools. Two evaluators found thinking about the abstraction gradient to be particularly confusing, but a third evaluator found this to be particularly useful; we believe that this heuristic was stated confusingly, and will be more useful once it is re-worded.

There was only one evaluator who did not strongly agree that using heuristics both helped them find new problems and gave them a new perspective. This remaining evaluator strongly agreed that the heuristic evaluation was a helpful

exercise, but thought the process clarified their thinking rather than providing an entirely new perspective. All evaluators thought heuristic evaluation would be a useful methodology for their own research.

6. Future Work

Although this paper provides initial compelling results on the use of heuristic evaluation by language researchers to find interesting issues with parallel language features, a rigorous evaluation of the heuristics presented in this paper remains an area of future work. In particular, we would like to run a larger survey in which the heuristics presented here are rated and expanded. We also would like to collect a set of usability problems for a variety of parallel language features in order to ensure that the heuristic set covers a great variety of potential problems.

Another goal is to further explore how other discount usability methods, such as cognitive walkthroughs, can be applied to programming languages features or concepts. Lastly, we would like to better understand adoption barriers within the programming languages research community which prevent similar methods from being used.

Acknowledgments

We would like to thank the evaluators for their participation and the anonymous reviewers for providing useful feedback.

References

[1] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, et al. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, 2009.

[2] A. Blackwell and T. Green. A Cognitive Dimensions questionnaire optimised for users. In *Annual Meeting of the Psychology of Programming Interest Group*, volume 12, pages 137–152, 2000.

[3] T. Buxton, A. Tarrell, and A. Fruhling. Heuristic Evaluation of Mission-Critical Software Using a Large Team. *International Conference on Human-Computer Interaction*, pages 673–682, 2009.

[4] S. Clarke. Evaluating a new programming language. In *Workshop of the Psychology of Programming Interest Group*, pages 275–289, 2001.

[5] J. Dagit, J. Lawrance, C. Neumann, M. Burnett, R. Metoyer, and S. Adams. Using cognitive dimensions: advice from the trenches. *Journal of Visual Languages & Computing*, 17(4):302–327, 2006.

[6] H. Desurvire, M. Caplan, and J. Toth. Using heuristics to evaluate the playability of games. In *Conference on Human factors In computing systems (CHI)*, 2004. Extended Abstract.

[7] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[8] W. Gray. Who Ya Gonna Call? You're on Your Own. *IEEE Software*, 14(4):26, 1997.

[9] T. Green and M. Petre. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.

[10] R. Jeffries, J. Miller, C. Wharton, and K. Uyeda. User interface evaluation in the real world: a comparison of four techniques. In *Conference on Human factors In computing systems (CHI)*, pages 119–124. ACM, 1991.

[11] S. Jones, A. Blackwell, and M. Burnett. A user-centred approach to functions in Excel. In *International Conference on Functional Programming (ICFP)*. ACM, 2003.

[12] D. Kane. Finding a place for discount usability engineering in agile development: throwing down the gauntlet. In *Agile Development Conference (ADC)*, 2003.

[13] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *SIGPLAN Notices*, 43(3):329–339, 2008.

[14] J. Mankoff, A. Dey, G. Hsieh, J. Kientz, S. Lederer, and M. Ames. Heuristic evaluation of ambient displays. In *Conference on Human factors In computing systems (CHI)*, 2003.

[15] J. Nielsen. *Usability Inspection Methods*. Wiley, 1994.

[16] J. Nielsen. Applying discount usability engineering. *IEEE Software*, 12(1):98–100, 1995.

[17] J. Nielsen and R. Molich. Heuristic evaluation of user interfaces. In *Conference on Human factors In computing systems (CHI)*. ACM, 1990.

[18] J. Pane and B. Myers. Usability issues in the design of novice programming systems. *Human-Computer Interaction Institute Technical Report CMU-HCII-96-101*, 1996.

[19] J. Pane, B. Myers, and L. Miller. Using HCI techniques to design a more usable programming system. In *Symposium on Human Centric Computing Languages and Environments*, 2002.

[20] D. Pinelle, N. Wong, and T. Stach. Heuristic evaluation for games: usability principles for video game design. In *Conference on Human factors In computing systems (CHI)*, pages 1453–1462. ACM, 2008.

[21] G. Sim, J. Read, and G. Cockton. Evidence based design of heuristics for computer assisted assessment. *Human-Computer Interaction (INTERACT)*, 2009.

[22] J. Yi and C. Flanagan. Effects for cooperable and serializable threads. In *Workshop on Types in Language Design and Implementation (TLDI)*, 2010.

[23] J. Yi, C. Sadowski, and C. Flanagan. Cooperative reasoning for preemptive execution. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2011.