"Automatically Assessing Code Understandability" Reanalyzed: Combined Metrics Matter

Asher Trockman,[†] Keenen Cates,[†] Mark Mozina,[†] Tuan Nguyen,[†] Christian Kästner,[‡] Bogdan Vasilescu[‡]
[†]University of Evansville, USA [‡]Carnegie Mellon University, USA

ABSTRACT

Previous research shows that developers spend most of their time understanding code. Despite the importance of code understandability for maintenance-related activities, an objective measure of it remains an elusive goal. Recently, Scalabrino *et al.* reported on an experiment with 46 Java developers designed to evaluate metrics for code understandability. The authors collected and analyzed data on more than a hundred features describing the code snippets, the developers' experience, and the developers' performance on a quiz designed to assess understanding. They concluded that none of the metrics considered can individually capture understandability. Expecting that understandability is actually captured by a combination of multiple features, we present a reanalysis of the data from the Scalabrino *et al.* study, in which we use different statistical modeling techniques. Our models suggest that some computed features of code, such as those arising from syntactic structure and documentation, have a small but significant correlation with understandability. Further, we construct a binary classifier of understandability based on various interpretable code features, which has a small amount of discriminating power. Our encouraging results, based on a small data set, suggest that a useful metric of understandability could feasibly be created, but more data is needed.

1 INTRODUCTION

Code understandability is critical for maintaining and developing software. Tasks such as creating new features, fixing bugs, code review, and refactoring require a deep understanding of code. Indeed, previous research shows that developers spend around 70% of their time understanding code [12]. An automatic measurement of code understandability could be used as a fitness function in refactoring tools, helping developers to write understandable code. It could also be used for quality assurance as part of a build system, as a heuristic to predict defects, or to assist open source project maintainers in assessing the quality of large numbers of pull requests.

Despite its value, such a metric remains an elusive goal. Previous attempts either have not been empirically evaluated [10, 11, 17], include a metric in a larger quality model [1, 3], or only measure understandability at the whole-system level [5]. Intuitively, we might expect that code complexity or readability are related to developers' ability to understand code, but there is a lack of empirical evidence to support this belief. Research on code comprehension tends to be focused on the cognitive processes of understanding code [18, 19], or common beliefs, rather than empirical evaluation.

Numerous models and metrics have been proposed for a related concept, code readability [4, 6, 7, 11, 13, 16]. Such metrics are generally based on developers' perceived readability scores of code snippets. For example, Buse and Weimer [4] trained an effective binary classifier for code readability based on a combination of many code metrics, using a data set of 100 small snippets labeled by 120 human subjects. However, readability scores do not necessarily reflect understandability: a code snippet may be readable, *e.g.*, following good naming and syntax conventions, but not understandable, *e.g.*, using complex logic or an unknown or poorly documented API.

The only exception to the literature scarcity on metrics for code understandability is the recent work by Scalabrino *et al.* [15]. In *"Automatically Assessing Code Understandability: How Far Are We?"* (hereafter "the original study"), the authors gathered a small data set on code understandability by having 46 students and developers evaluate Java code snippets. Importantly, they measured *actual understandability* with three quiz questions for each snippet evaluated. This diverges from previous research on readability, which is based only on perception. For each snippet, they provide 121 metrics based on code, documentation, and developer experience. These metrics were individually analyzed for correlation with understandability, and the study concluded that no individual metric was significantly correlated with the outcome.

In this paper, we question Scalabrino *et al.*'s fundamental assumption that *understandability can be captured with a single metric*, and conjecture, much like Buse and Weimer in their model of readability [4], that *understandability is better described by a combination of metrics.* To test our conjecture, we perform a reanalysis of the data in the original study, but instead use *multivariate* statistical modeling techniques for inference and prediction, as opposed to the analysis of pairwise correlations in the original study. Specifically, we investigate the following research questions: **RQ1:** *Do any of the 121 metrics recorded correlate with understandability, other variables held equal?* and **RQ2:** *Is it possible to create a measure of code understandability based on a combination of these 121 metrics?*

Our study can be considered a *reanalysis*, since we are applying different statistical methods to the same data set; we did not repeat the human evaluation of code snippets. More loosely, this could be considered a specific type of replication, known as "pseudoreplication", "complete secondary analysis", or "internal replication" [9]. Reanalysis and replication in general is invaluable to the integrity of empirical software engineering research, and we thank Scalabrino *et al.* for making their data set publicly available for this purpose.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

MSR '18, May 28–29, 2018, Gothenburg, Sweden

© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.

ACM ISBN 978-1-4503-5716-6/18/05…$15.00
https://doi.org/10.1145/3196398.3196441

2 DATA SET

Collection. In the original study,[1] 46 developers of varying levels of experience were given a quiz to assess understanding of Java code snippets. The code was taken from various large open source projects such as Spring Roo and Weka. The quiz was implemented as a web application that presented multiple questions. Each question consisted of a snippet of code, for which a developer had unlimited time to read and search online, followed by a yes/no response on whether they understood the code (perceived understandability). Three verification questions were then asked to see if the code was truly understood (actual understandability). These were multiple choice questions, each with four choices, which primarily focused on the purpose (intent) of variables and methods. Developers were asked to complete the quiz on eight separate code snippets. Of the 46 developers, 39 answered all eight questions, while 7 answered between 1 and 3 questions, for a total of 324 observations.

Along with the developers' responses, 121 features about the code snippet and the developers' experience were recorded. The developers' years of experience programming in general and in Java specifically (Fig. 1) were recorded, in addition to the their position (BS/MS/PhD student, professional developer, open source developer). It's clear from Fig. 1 that years of experience in Java is correlated with the percentage of verification questions answered correctly.

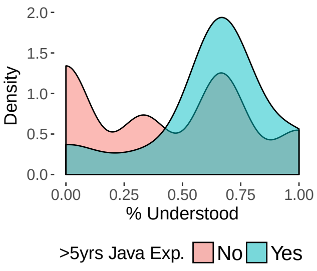

Figure 1: Java experience

Most of these features were computed from the syntactic structure of the code, which was hypothesized to relate to understandability: the number of lines, the number of tokens, the number of variables, *etc.* Some features were computed from the documentation quality of the code, *e.g.*, prevalence of comments or of internal documentation for the methods used.

Processing. In preparation for multivariate regression modeling, and following the principle of parsimony (seeking simple, more generalizable models), we made several data transformations. First, our intuition is that the quality of being a student should be less important than the amount of programming experience; given also the high class imbalance (32 of the participants were BS students), we exclude this variable from our analysis. Instead, we introduce two new variables based on self-reported years of programming experience. We call a participant a *professional* if they have five or more years of experience programming in any language, and we call them a *Java professional* if they have five or more years of experience specifically in Java. In this case, 13 of the participants were professionals, and 4 were Java professionals. Although the classes are imbalanced, we expect that Java professionals have a significant advantage in answering the questions; therefore, this is an important variable to include in our models.

Second, the *actual understandability* scores take on values in $\{0, 0.334, 0.667, 1.0\}$, which we could potentially model using linear regression or multinomial logistic regression. However, to facilitate model selection and interpretation, we assert that a developer understood the code if they answered more than half of the verification questions correctly. Hence, we introduce a new binary response variable, *understood*, to capture this distinction.

Otherwise, we do not remove any entries (the original authors removed outliers already),and use the variance inflation factor (VIF) with a cutoff of 3 to diagnose multicollinearity [2].

3 ANALYSIS

We hypothesize that professional developers, by our operationalization, are more likely to understand the code snippets. Beyond this, we approach our analysis with no *a priori* hypotheses, and use model selection techniques to guide our exploration of the data.

What explains understandability? We construct binary logistic regression models to see if a combination of metrics about code, documentation, and developer experience can *explain* actual understandability. Since each participant answered up to eight questions, the samples are not independent; each participant has a different amount of experience and knowledge. To address this, we use mixed-effects models [?], with a random effect to capture each participant's baseline ability to understand the code.

To find such a combination of metrics, we proceed with two methods: First, we use principal component analysis to find a "small number of interesting dimensions" [8], by automatically grouping together correlated metrics and revealing which metrics explain most of the variability. Based on the principal components, we manually select "interesting" and minimally correlated metrics to include in our model. Second, we use forward stepwise selection [8], where metrics are added to the model according to an information criterion (AIC) [8]. This will also help to identify potentially interesting metrics for further consideration. The following describes each of the two methods in detail.

Principal component analysis. To inform our exploration, we did principal components analysis [8] to identify which metrics explain the most variance in the data set. The metrics were automatically scaled by R's PCA function. We found that there is not a specific principal component that explains most of the variance (Fig. 2). This is corroborated by the original study, which found that no individual metric was strongly correlated with understanding. For simplicity, we decided to analyze the loadings of the first six components, explaining approximately 55% of the variance.

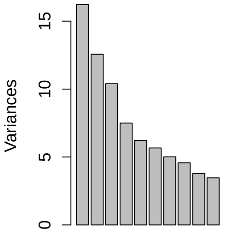

Figure 2: PCA Scree Plot

By manual inspection, we found that the first six components were roughly correlated with cyclomatic complexity, Halstead's volume, lines of code, number of operators, maximum line length, and number of spaces. In some cases, the data set provides several related metrics, *e.g.*, about indentation and spacing. We chose the most familiar metric in each case, *e.g.*, cyclomatic complexity instead of the "discrete Fourier transform of conditionals" [7]. Additionally, we add the binary variable *professional* to better control for participant experience.

[1] The data set is available online at https://dibt.unimol.it/report/understandability/

Table 1: Explanatory models of understandability

(a) Model based on manual selection and PCA.

Variable	Coefficient
(Intercept)	$-0.329^{\cdot}$
Professional	1.11^{***}
Cyclo.complexity	-0.044
LOC	-0.144
Volume	-0.033
X.operators..avg.	-0.018
X.spaces..avg.	-0.158
Line.length..max.	-0.392^{**}
N	324
Log Likelihood	-198.174
AIC	414.349
BIC	448.376
R^2_m	0.0991
R^2_c	0.2061

$^{***}p < .01$; $^{**}p < .05$; $^{*}p < .1$

(b) Stepwise selection mixed effects model.

Variable	Coefficient
(Intercept)	-0.398^{*}
Indentation.length..dft.	0.241
Literals..Visual.Y.	-0.275^{*}
JavaProfessional	1.228
X.parameters	-0.307^{*}
NMI..avg.	0.442^{***}
MIDQ..max.	-0.392^{**}
X.periods..avg.	-0.325^{**}
Professional	1.014^{**}
Line.length..max.	-0.488^{***}
TC..avg.	-0.793^{***}
N	324
Log Likelihood	-185.00
AIC	394.00
BIC	439.37
R^2_m	0.2870
R^2_c	0.4106

$^{***}p < .01$; $^{**}p < .05$; $^{*}p < .1$

The model, summarized in Table 1a, fits poorly, having a pseudo-R^2 (R^2_c) of about 20%.[2] The only variables with a significant effect are *max. line length* and *professional.* This is consistent with the original study, which reported *max. line length* as one of the metrics most correlated with understanding, though still to a small extent.

Interpreting the coefficients from the model, we note that: (1) being a professional programmer increases the odds of understanding the code by a factor of 3; and (2) the maximum line length increasing by one character decreases the odds by a factor of 0.7.

Stepwise selection. We conclude that understandability cannot be a function of line length only, so we proceed to refine our variable selection process. First, we removed twelve variables with missing values. Next, to determine which metrics to include in the model, we implemented a forward stepwise selection algorithm based on AIC [8], iteratively selecting the metric which most reduces the AIC. We selected ten variables and checked for multicollinearity issues (VIF below 3). Since stepwise selection methods can bias significance levels [**?**], we constructed a bootstrap distribution of each coefficient in the model for additional validation, as per [**?**].

The resulting model is summarized in Table 1b. As expected, the new model is better at explaining understandability than the previous model, with an R^2_c of 41%, and nearly all of the explanatory variables have significant effects.

We performed model diagnostics, removing eight high influence points with Cook's distance greater than $\frac{4}{n}$. Upon inspection, it appears most of these points were *professional* Java programmers who answered incorrectly. The data set also includes the time taken for a developer to answer a question (not analyzed); in the case of the most influential points, the time taken was 0 seconds, implying the question was perhaps not even read. The model fit to the reduced data set was not significantly different than that fit to the original, so we do not provide the parameters for the former.

Interpreting the most significant coefficients ($p < 0.01$) in Table 1b, we draw the following observations: (1) A one unit increase in *NMI.avg.*, or *narrow meaning identifiers*, a measure of the descriptiveness of variable names [14], increases the odds of understanding by a factor of 1.55; this appeals to common beliefs. (2) A unit increase in *MIDQ.max.*, or *methods internal documentation quality*, a measure of the quality of documentation of the methods called in the code snippet [15] (*e.g.*, from other libraries), correlates with lower understandability, or a change by a factor of 0.67. We find it unintuitive that an operationalization of better documentation correlates with lower chances of understanding. We conjecture that more documentation could be correlated with more complex code. (3) A larger number of periods per line, *X.periods.avg.*, is correlated with lower understandability. Since periods are associated with calling methods from other classes, this agrees with intuition. (4) As we would expect, being a Professional increases the odds of understanding by a factor of 2.75. (5) As in the previous model inspired by PCA, we find that an increase in *max. line length* correlates with lower odds of understandability. (6) Average textual coherence (*TC..avg.*), a metric proposed as a measure of the "closeness of concepts" in a method [14], is counterintuitively associated with much lower understandability by a multiplicative factor of 0.45.

Can we predict understandability? The previous *explanatory* models attempt to identify the factors that correlate with understandability based on the whole data set. We now construct a model that can make predictions based on new, unseen observations.

Since random effects cannot be estimated based on a single new observation, we instead fit a fixed effects binary logistic regression model with understandability as a response. Such a model, given the independent variables, predicts a value between 0 and 1: generally a prediction greater than the threshold 0.5 is taken to be the positive class (*understood*). This value could be interpreted as a metric of *percentage understanability.*

We use LASSO (Least Absolute Shrinkage and Selection Operator) regression [8], which is similar to least-squares regression, but has the added benefit of automatically selecting a subset of the metrics to be used in the model, which prevents overfitting; that is, if we included all the metrics, the model would probably have good performance on our data set but bad performance on new data. Since most of our 121 variables will be eliminated, this also results in a more interpretable model that we think would be appropriate for an automatic metric of understandability: specific issues, such as indentation or complexity, could be pointed out.

LASSO requires a tuning parameter, which we selected according to multiple runs of 10-fold cross validation [8]. We scaled metrics to be on the same scale and converted factors to dummy variables. To simulate new observations, we evaluated the model by splitting the data into a training set (75% of the data) and a test set (25%). The model was trained with the training set and its predictive performance evaluated on the test set. Since it is possible to be "lucky", we averaged our results over 5000 random splits, finding a new tuning parameter on the training data each time.

A common way to evaluate classifier performance is with an ROC curve (example in Fig. 3). This plots the "true positive" versus

[2] Since we use models with random effects, we compute pseudo-R^2 values to assess goodness of fit: one without random effects, R^2_m, and one with, R^2_c [**?**].

the "false positive" classification rate for various thresholds (we suggested 0.5 in the first paragraph). Importantly, an ROC curve that follows the diagonal line implies that the model has no predictive power. An optimal curve would hug the top left corner. The performance of the classifier over all thresholds can be summarized by the area under the curve (AUC) [8]; an optimal value is 1.0, while 0.5 implies guessing.

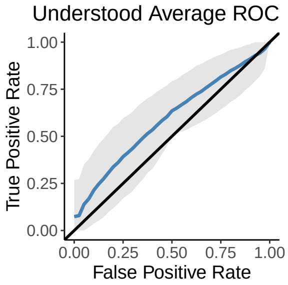

Figure 3: ROC curve for LASSO model

Fig. 3 displays the average ROC curve for each of the 5000 random train/test splits in blue, and a 95 percentile band in gray. On average, the curve falls above the diagonal; that is, the model has a small amount of discriminating power. The average AUC is 0.59, while the 2.5%-tile and 97.5%-tile are 0.46 and 0.70, respectively.

Since the LASSO does not select the same subset of metrics for each random split [8], we report some commonly occuring metrics: max. line length (95%), professional (80%), Java professional (73%), narrow meaning identifiers (73%), number of parameters (81%), number of numbers (61%), textual coherence (53%), number of periods (53%), methods internal documentation quality (60%). The average number of metrics selected was 15. Overall, these agree with the metrics selected for explanation earlier with stepwise selection.

4 DISCUSSION AND CONCLUSION

Research questions. Overall, it seems that *combined metrics matter*: our models can explain some of the underlying variability in understanding. Most of our results appeal to intuition: more specific variables, fewer parameters and literals, fewer periods, and shorter lines are correlated with more readable code. However, we did not expect to find that documentation quality and textual coherence would be associated with less understandable code.

At the least, we can answer **RQ1** positively: various computed characteristics of the code snippets correlate with their understandability, other variables held equal. In both the PCA inspired model and the stepwise selected model, it is clear that the greatest factor in code understanding is the participant's level of experience; yet, our models show that even when this is controlled for, other metrics still play a significant role in understandability, a conclusion that could not be achieved with the analysis of bivariate correlations used in the original study.

RQ2 asked whether we could capture a metric of understandability using a combination of the 121 metrics considered. After performing LASSO regression with cross-validation, we found a small amount of evidence that it is possible to discriminate hard-to-understand code from easy-to-understand code. While an AUC of 0.5 is equivalent to guessing understandability, our model achieved an average AUC of 0.59, which implies some discriminating power. Our proposed metric would be the logistic regression prediction which falls between 0 and 1, and could be interpreted as *percent understandability*. To be useful, this functionality would require a high level of accuracy, which our model cannot provide. However, in light of our model's small amount of predictive power, we expect that accuracy could be increased with a larger data set provided by a more comprehensive human study of code understanding.

Unlike the original study, which only considered the correlation between individual metrics and understanding, we combined metrics in statistical models, expecting that the accumulation of many different factors contributes to understandability. We found a small amount of evidence that such combinations can explain understandability better than individual metrics, a more positive conclusion than the original study.

Limitations & threats to validity. The greatest limitation of this study was the relatively small data set; the original study provided only 324 observations from 46 developers. Such data sets require a large amount of effort from both the experimenter and the participants, thus it is difficult to get larger amounts of similar data.

Given such high dimensionality (121 explanatory variables), it is possible for correlations to be spurious. We attempted to alleviate this possibility by constructing bootstrap samples of model coefficients and verifying that the produced confidence intervals do not contain zero (explanatory section) and by using training/testing sets for validation (predictive section). Nonetheless, such concerns are inherent to exploratory data analyses and cannot be avoided without collecting more data in a new experiment.

We used forward stepwise selection based on AIC to produce a model for inference, a practice that is controversial and subject to ongoing research [**?**], since it can result in inflated significance. However, we limited the model to ten predictors to reduce overfitting and verified that the model performs well on bootstrap samples [**?**]. The selected predictors mostly agree with our intuition about code understanding. Note that the model based on PCA is not vulnerable to such inflation.

In some instances, we found that the verification questions used in the original study were excessively shallow, *i.e.*, they did not agree with our notions of understanding. Many focused on the purpose of specific variables, ignoring the "big picture", and some questions had ambiguous answers differing by only a single word. In constructing a metric of understandability, we think that a large, high-quality data set should be the foremost concern.

Implications for developers. Though we were not able to present a model that assessed code understandability with high accuracy, given our limited success with a small data set and relatively simple modeling techniques, we expect that understandability will be measureable in the future.

We used conventional statistical techniques to predict understandability, allowing for easy interpretation, *i.e.*, determining specific features of code that reduce understandability. For maximimal usefulness, future work should continue to embrace principle of parsimony, favoring the simplest possible model.

As more effective measures of understandability are developed, IDE plugins could be offered which notify the user of the quality of the code they have written. Given interpretable models, specific suggestions could be given and the user could view their improvements. Distributed projects, those with many disparate developers, could include understandability thresholds in their build systems, automatically showing project maintainers a summary of understandability before taking valuable time to do manual code review.

Ultimately a metric of understandability would make software engineering more efficient, reducing costs and the prevalence of faults.

REFERENCES

[1] Krishan K Aggarwal, Yogesh Singh, and Jitender Kumar Chhabra. 2002. An integrated measure of software maintainability. In *Reliability and maintainability symposium, 2002. Proceedings. Annual*. IEEE, 235–241.

[2] Paul D Allison. 1999. *Multiple regression: A primer*. Pine Forge Press.

[3] Marc Bartsch and Rachel Harrison. 2008. An exploratory study of the effect of aspect-oriented programming on maintainability. *Software Quality Journal* 16, 1 (2008), 23–44.

[4] Raymond PL Buse and Westley R Weimer. 2010. Learning a metric for code readability. *IEEE Transactions on Software Engineering* 36, 4 (2010), 546–558.

[5] Andrea Capiluppi, Maurizio Morisio, and Patricia Lago. 2004. Evolution of understandability in oss projects. In *Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on*. IEEE, 58–66.

[6] Ermira Daka, José Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer. 2015. Modeling readability to improve unit tests. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 107–118.

[7] Jonathan Dorn. 2012. A general software readability model. *MCS Thesis available from (http://www. cs. virginia. edu/˜ weimer/students/dorn-mcs-paper. pdf)* (2012).

[8] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2017. *An Introduction to Statistical Learning with Applications in R*. Springer.

[9] Natalia Juristo and Omar S Gómez. 2012. Replication of software engineering experiments. In *Empirical software engineering and verification*. Springer, 60–88.

[10] Jin-Cherng Lin and Kuo-Chiang Wu. 2006. A model for measuring software understandability. In *Computer and Information Technology, 2006. CIT’06. The Sixth IEEE International Conference on*. IEEE, 192–192.

[11] Jin-Cherng Lin and Kuo-Chiang Wu. 2008. Evaluation of software understandability based on fuzzy matrix. In *Fuzzy Systems, 2008. FUZZ-IEEE 2008.(IEEE World Congress on Computational Intelligence). IEEE International Conference on*. IEEE, 887–892.

[12] Roberto Minelli, Andrea Mocci, and Michele Lanza. 2015. I know what you did last summer: an investigation of how developers spend their time. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*. IEEE Press, 25–35.

[13] Daryl Posnett, Abram Hindle, and Premkumar Devanbu. 2011. A simpler model of software readability. In *Proceedings of the 8th working conference on mining software repositories*. ACM, 73–82.

[14] Simone Scalabrino, Gabriele Bavota, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. 2016. Improving code readability models with textual features. In *Proceedings of the 24th IEEE Int’l Conference on Program Comprehension (ICPC 2016)*. IEEE Press, Piscataway, NJ, USA.

[15] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. 2017. Automatically Assessing Code Understandability: How Far Are We?. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 11.

[16] Simone Scalabrino, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. 2016. Improving code readability models with textual features. In *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*. IEEE, 1–10.

[17] D Srinivasulu, Adepu Sridhar, and Durga Prasad Mohapatra. 2014. Evaluation of Software Understandability Using Rough Sets. In *Intelligent Computing, Networking, and Informatics*. Springer, 939–946.

[18] M-A Storey. 2005. Theories, methods and tools in program comprehension: Past, present and future. In *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*. IEEE, 181–191.

[19] M-AD Storey, Kenny Wong, and Hausi A Müller. 2000. How do program understanding tools affect how programmers understand programs? *Science of Computer Programming* 36, 2-3 (2000), 183–207.