Reducing the State Space of Programming Problems through Data-Driven Feature Detection*

Rui Zhi
NC State University
rzhi@ncsu.edu

Thomas W. Price
NC State University
twprice@ncsu.edu

Nicholas Lytle
NC State University
nalytle@ncsu.edu

Yihuan Dong
NC State University
ydong2@ncsu.edu

Tiffany Barnes
NC State University
tmbarnes@ncsu.edu

ABSTRACT

The large state space of programming problems makes providing adaptive support in intelligent tutoring systems (ITSs) difficult. Reducing the state space size could allow for more interpretable analysis of student progress as well as easier integration of data-driven support. Using data collected from a CS0 course, we present a procedure for defining a small but meaningful programming state space based on the presence or absence of features of correct solution code. We present a procedure to create these features using a panel of human experts, as well as a data-driven method to derive them automatically. We compare the expert and data-driven features, the resulting state spaces, and how students progress through them. We show that both approaches dramatically reduce the state-space compared to traditional code-states and that the data-driven features have high overlap with the expert features. We conclude by discussing how this feature-state space provides a useful platform for integrating data-driven support methods into ITSs.

Keywords

Programming, ITS, Intelligent Support, State-space

1. INTRODUCTION AND MOTIVATION

Analyzing programming data often requires a way to represent a student's current *state* on a given problem. A good state representation should reflect a student's progress towards a correct solution, while allowing researchers to identify when students meaningfully *change* to a new state and when two different students are in the *same* state. This enables a data-driven system, such as an intelligent tutoring system (ITS), to represent how a group of students traverse through the *state space* when problem solving, (e.g. with an interaction network [2]), in order to provide adaptive support, such as data-driven hints [7, 8] or worked examples [5]. In the domain of programming, a student's state is typically represented by their current code, called a *code-state*. However, this representation leads to very large and poorly connected state spaces [6, 8], which makes it difficult to compare students and apply data-driven methods. To address this, Rivers and Koedinger showed that a code-state space size can be reduced up to 41% by applying canonicalization to reduce the *syntactic* variability of the code while maintaining its *semantic* meaning [8]. Similarly, Peddycord et al. [4] presented a *world-state* representation, defined by the output of a student's code, rather than the code itself, greatly reducing the state space. However, canonicalization may be insufficient on larger or more open-ended problems [6], and world-states may not contain enough information to successfully adapt data-driven support.

In this work, we present a *feature-state* representation that defines a student's state by the presence or absence of specific features of a correct solution. We demonstrate that this representation dramatically reduces the size of an open-ended programming state space, while creating semantically meaningful states. We present two methods for defining features, using experts and a data-driven algorithm, and show these methods can produce comparable results. Finally, we show that feature-states can be used to visualize student solution paths, and we argue that they can provide a representation for generating and tailoring data-driven support.

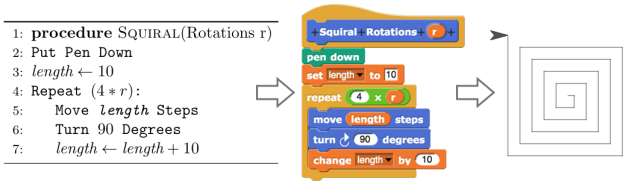

Figure 1: Squiral pseudocode, Snap*!* code and output.

2. FEATURE-SPACE CREATION

We explored state space reduction in the context of a CS0 course for non-CS-majors using datasets collected during the Spring 2016 (S16), Fall 2016 (F16) and Spring 2017 (S17) semesters[1]. Students completed 6 programming assignments in iSnap [7], a block-based programming environment based on Snap*!* [3], where students create 2D graphical output with sprites. iSnap offers students on-demand, data-driven hints, but these hints may alter students' problem-solving patterns. We therefore excluded students who requested hints, and our remaining data contained code traces from 24, 38, and 29 students for the S16, F16, and S17 semesters, respectively. Each *code trace* consisted of a sequence of timestamped *snapshots* of student code. We chose one assignment, Squiral, to explore in-depth where students must write a procedure to draw a spiraling square-like shape. Common

*This material is based upon work supported by the National Science Foundation under grant 1623470.

[1]Datasets available at go.ncsu.edu/isnap

Feature Name	Brief Description	Data-driven Analogue	Similarity
E1. Procedure	Primary code inside of a procedure.	D1: Create a procedure OR a variable.	0.761
E2. Draw Anything	Able to draw anything on screen.	D8: Use a 'repeat' AND create a variable OR a parameter.	0.737
E3. Move 'Square -like'	Able to move sprite in a square-like fashion.	D11: Have a 'move' AND a 'turn' in a 'repeat' AND have a 'pen down.'	0.648
E4. Correctly Use Parameter	Correctly uses parameter within custom block.	D4: Have a 'repeat' inside a procedure.	0.671
E5. Repeat Correct # of Times	Repeats square-like movement correct number of times.	D5: Have a 'multiply' block with a variable OR two nested 'repeats'.	0.711
E6. Move 'Variably'	Movement is based on a variable not literal amount.	D10: Have a 'move' with a variable argument inside of a 'repeat'.	0.579
E7. Move 'Squirally'	Increase length to move for each side.	D7: Change a variable inside a 'repeat'.	0.610

Table 1: Expert Features and Corresponding Data-Driven Features with highest Jaccard Similarity

solutions contain 7-10 lines of code and use a procedure, loops, variables, and arithmetic operators (see Figure 1).

In this work, we hypothesize that the state of a student on a given assignment can be meaningfully defined by a set of code *features*. Each feature should describe a distinct property of correct solution code. A feature can specify a necessary code structure (e.g. in Squiral, a student must write their code within a procedure) or required program output (e.g. in Squiral, running code must draw something to the screen). Table 1 gives further examples of features for the Squiral assignment. Features are different from unit tests because they can be present in incomplete or unrunnable code, and they are different than skills or knowledge components because features describe assignment-specific properties of the code, not student ability. We define the presence or absence of each feature in a student's code as a student's *feature-state*. These feature-states reduce the state space to a finite size (the power set of features), while capturing a student's progress towards a solution state (where all features are present). We present two methods for creating a set of features for Squiral, one using human experts and one using a data-driven approach. We show graph characteristics for both resulting state spaces which demonstrate the desired effect of reducing the state space size and creating a stronger connected graph with interpretable student paths.

2.1 Expert Features

Three experts in Snap*!* and the Squiral assignment were tasked with generating features of correct solutions using student submissions from the largest dataset (F16). Each expert independently reviewed 10 out of an initial set of 15 traces, and used these to develop a preliminary set of feature of correct solutions. They then convened to resolve their features into a final set of 7 features (described in Table 1). A codebook was developed to determine, for any given snapshot in a student's trace, whether each feature is present. Each expert then independently tagged a shared set of 10 traces with whether each feature was present or absent in each snapshot (defining a feature-state for that snapshot). After an initial round of this procedure, the codebook definitions were revised and the procedure was redone resulting in inter-rater agreement with Fleiss' kappa between 0.65 and 0.98 for all 7 features. Experts then divided the remaining traces and independently tagged their snapshots with feature-states, to create a total of 38 tagged traces. Finally, a smoothing function was applied to the tagged traces in which periods of rapid feature-state changes (defined as transitioning back and forth between two feature-states within a 5 snapshot window) were set to the values of the subsequent period after the variance. This was necessary to reduce noise generated by actions in Snap*!* that do not represent meaningful feature-state changes (e.g. when a student "picks up" code blocks, temporarily removing them, and then replaces them). The final result was a feature-space developed from the 38 traces comprising of 6,339 tagged snapshots.

Attributes of the expert *feature-space* are reported in Table 2, alongside the original code-state space (with basic canonicalization) and the data-driven feature-space (explained in Section 2.2). While the total possible number of feature-states is 128 (2^7), only 38 were observed, and 31 traversed, by multiple student traces. This was due to both explicit dependencies (i.e. features that are defined as requiring other features to be present) and implicit dependencies in feature completion (i.e. students almost always do some features before others). The observed feature-state space is not only absolutely smaller than the observed code-state space ($\sim$1% of the size), but the proportion of unique states (traversed by only one trace) is much smaller in the expert feature-space (18%) than the code-state space (99.6%), suggesting that the feature-states effectively collapse similar student code into single states.

In Figure 2, we visualize student completion of expert features by showing the order in which students accomplished these features. The thick green lines indicate that more than 10% of traces traversed that edge, and the state's size reflects the number of students who visited that state. Over 50% of students began by completing E1 (creating a procedure), and most others completed E2 (drawing something) first. Most students then created a procedure parameter (E4) and made their sprite move 'square-like' (E3), and nearly all ended by completing E5 (repeat the correct # of times) or E7 (move 'squirally'). Stuck states, where a student submits an incomplete assignment, are shown as octagons in which the shaded portion matches the portion of students who did not progress. The concentration of stuck states towards the beginning (columns 2 and 3) show students getting stuck on completing E3 (move 'square-like') and E6 (moving using a

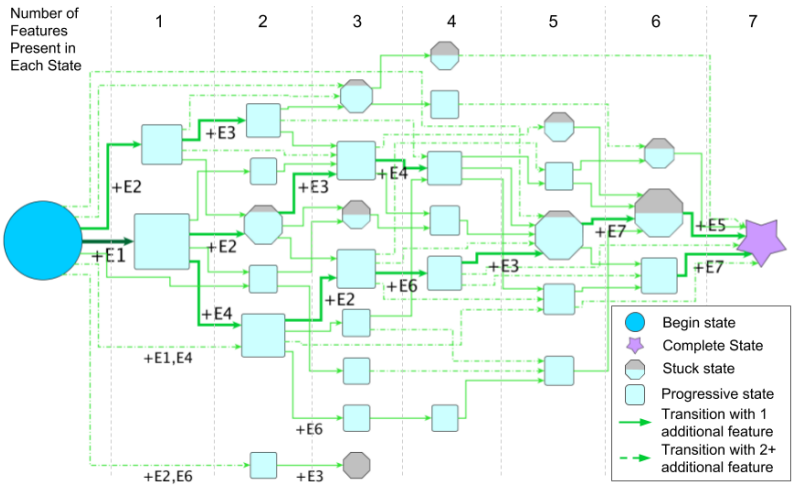

Figure 2: Student Progress Visualization on Expert Features

F16 (n=38)	CS	EF	DDF
# Features	–	7	11
Possible States	$\aleph_0$	128	1028
# States Observed	3069	38	169
% Unique	99.4%	18%	65%
# Edges	3609	155	283
% Unique	99.8%	68%	83%

Table 2: Attributes of three state spaces: code-states (CS), expert feature-states (EF), and data-driven feature-states (DDF). "% Unique" is how many nodes/edges were observed in only one student's trace.

variable), while the large stuck states in columns 5 and 6 demonstrate students' difficulty completing E5.

2.2 Data-Driven Features

While our expert features reduce the programming state space well, they are time-consuming to produce. We hypothesized these features were closely tied to structural properties of the student's abstract syntax tree (AST), so we developed a data-driven procedure to extract features automatically. Our data-driven features are built using *code shapes*, common subtrees of correct student ASTs. The presence or absence of these code shapes should indicate how close a student is to a correct solution. Specifically, we used pq-Grams [1] as our code shapes, though any subtree could be used. These consist of a single node, its $(p-1)$ immediate ancestors and q of its children (or blanks when no children are present). These pg-Grams can represent diverse code, and they have been used in data-driven hint generation [9].

Given a set of student traces T for an assignment, we generate data-driven features as follows: We identify an initial set of code shapes, C, present in *any correct* solution in T. We add all observed pq-Grams to C for $p \in 1 \ldots 3$ and $q \in 1 \ldots 4$. Many of these code-spaces will be redundant, so we attempt to identify and remove duplicates by creating a set, S, consisting of *all* code snapshots from each attempt T. For each code shape $c \in C$, we create a subset $S_c \subseteq S$ consisting of all snapshots $s \in S$ that contain the code shape c. For each pair of code shapes $\{c_i, c_j\} \in C \times C$, we compute the Jaccard similarity as $|S_{ci} \cap S_{cj}|/|S_{ci} \cup S_{cj}|$ to determine how frequently the code shapes appear together. For any pair with Jaccard similarity over a threshold (we used 0.975^2), we consider these code shapes equivalent, and keep only the more descriptive code shape (i.e. with the larger, p and q).

We define the support for a code shape c as the proportion of correct solutions that contain c. Intuitively, only code shapes with very high support should be used in features. However, it is possible there are multiple strategies for completing a given problem which will yield different code shapes that only appear in one of the strategies. We therefore identify a set of *decision shapes*, where a decision d is a disjunction of code shapes $d = c_1 \vee \ldots \vee c_n$, which is contained by a solution if any of these code shapes are contained. Meaningful decisions contain mutually exclusive choices, so most correct solutions should contain just one of the decision's code shapes. To identify decisions, we define the overlap between code shapes c_i and c_j as the portion of correct solutions which contain c_j that *also* contain c_i. For each code shape $c_i \in C$, we iteratively identify the code shape $c_j \in C$ with the smallest overlap and less support than c_i and create a decision $d = c_i \vee c_j$. We add these decisions to C, and remove any code shape or decision shape where the support is below a given threshold (0.9^2 here).

C now defines a set of code and decision shapes that are contained in most solutions, similar to our expert-authored features. However, C may be much larger than the set of 7 features identified by our experts, which is undesirable, since the potential state space grows exponentially with each feature. We therefore combine our code and decision shapes to make features. A feature is a conjunction of shapes $f = c_1 \wedge \ldots \wedge c_n$, which is present only if each of its component shapes are present. We start with one feature $f_c = c$ for each $c \in C$ and then hierarchically cluster them, combining similar features together into larger conjunctions. To measure the goodness of any clustering of features, we define the *resolution* of a feature set as its ability to distinguish successive snapshots in a trace as being in different states. Our ideal resolution is given by the initial, full set of features, and resolution is lost whenever we combine two features. At each iteration of clustering, we combine the two most co-present features (highest Jaccard similarity). We choose the number of features using the "elbow method," identifying a number that comes before a steep drop in resolution.

We applied the procedure to the F16 iSnap dataset, which our human experts used to generate their features. The dataset contained 1,974 code shapes with a support above

[2]We chose conservative thresholds that worked well for this dataset, but they may not generalize to other datasets.

0.1, of which 233 (11.8%) were distinct, and 87 distinct decision shapes. Of these, 27 code shapes and 4 decision shapes (31 total) had a support above 0.9. We combined these into 11 features, which preserved 71% of the resolution of the original 31 shapes. As with the expert-authored features, we used these 11 data-driven features to define a feature-state space. We calculated the data-driven feature-state of each snapshot in the F16 dataset, applying the same smoothing function described in Section 2.1. Statistics on the state space defined by these features is also given in Table 2. While the data-driven feature-state space was larger than the expert feature-state space, due to a higher number of features, the data-driven space was still dramatically smaller in terms of number of states and edges and had better state and edge overlap than the basic code-state canonization.

3. COMPARISON AND DISCUSSION

To determine how well our data-driven features matched our expert features, we calculated the pairwise Jaccard similarity $J(e_i, d_j)$ between each expert feature $e_i \in E$ and each data-driven feature $d_j \in D$ over all snapshots in the F16 dataset. For each expert feature e_i, we identified the data-driven feature d_j with the highest relative similarity: $J(e_i, d_j) / \sum_{e \in E} J(e, d_j)$. The resulting data-driven analogues for each expert feature are shown in Table 1, along with their Jaccard similarity. Some pairings are very interpretable, such as E3 (move 'square-like') and D11, which requires a 'move' and 'turn' inside of a 'repeat'. However, D11 also requires a 'pen down,' so it also combines elements of E2 (draw anything). D5 is an example of an effective decision shape, identifying that students could repeat the correct number of times (E5) *either* by repeating ('sides' * 4) times or by nesting two repeats (one 'sides' times; one 4 times).

Only one of the 7 pairs, E2 (draw anything) and D8 (have a repeat and a variable or parameter), lack an obvious relationship. However, their relatively high Jaccard similarity (0.737) suggests that students were often using loops and variables when drawing. Some expert features share characteristics with multiple data-driven features like E4 (use a parameter correctly) which generally requires the presence of its analogue, D4 (have a 'repeat' inside of a procedure), since the procedure's parameter should be used in the repeat. However, it is also closely related to the unpaired data-driven feature, D9 (create a procedure with a parameter and have a main script). Another unpaired data-driven feature, D6 (call a custom procedure with an argument), was also considered by our experts as a possible feature, but they decided it was not meaningful enough for distinguishing a student's state. The presence of clear analogues for almost all expert features suggests that our data-driven procedure reflects human intuition of what code shapes are conceptually related.

3.1 Experiments

Two use-cases for our feature-space representation are determining when a student meaningfully transitions to a new state (within-student similarity) and determining when two students are in a similar state (between-students similarity). We investigated how well our data-driven features match the expert-authored features on these two tasks for the F16 dataset. Since in practice any data-driven features would need to be extracted ahead of time, we created a new set of 11 data-driven features using two *other* semesters as training data (S16 and S17, with $n = 53$ total traces).

To evaluate within-student similarity, from each trace in the F16 dataset ($n = 38$), we extracted all unique pairs of snapshots from the trace that occurred within 25 edits of each other (i.e. a window size of 25). For each snapshot pair, we calculated whether the snapshots were in the *same* state (all features match) or a *different* state, using both the expert and data-driven feature-spaces, and we calculated the agreement between them. The median agreement over all traces between our data-driven and expert features was 74.5% (IQR = 17.8%), with a median Cohen's kappa of 0.454 (IQR = 0.307), indicating moderate agreement. As a baseline for comparison, we also created a state space based on tree edit distance, which is commonly used to compare code snapshots [8, 9]. We sampled approximately 3,000 snapshots from the training dataset, drawing equally from each trace, and used k-medoids clustering to group them into 38 clusters, to match the 38 states produced by the expert features. When clustering, we defined the distance between two snapshots as the tree edit distance between their ASTs. We defined the state of a newly observed snapshot as the cluster number of the closest medoid. When determining if a pair of snapshots from the same student were in the same state, the distance-based features had a median agreement with the expert features of 68.8% (IQR = 17.6%), with a median kappa of 0.336 (IQR = 0.349), 74% as high as our data-driven features. A wilcoxon signed-rank test showed that the difference in kappa values was significant over all 38 traces ($W = 511$; $p = 0.041$). Different window sizes produced median kappas ranging from 0.336-0.489 for the data-driven features, but in each case they outperformed the edit distance approach. These results suggest our data-driven features can detect when students meaningfully change state, and the relatively high agreement with experts suggests that feature extraction could reasonably be automated.

To evaluate between-students similarity, we randomly sampled 25 between-student pairs of snapshots from each pair of traces in the F16 dataset, for a total of 17,575 pairs. We defined the feature-similarity between two snapshots as the proportion of features which were equivalent between the states (both present or both absent). We found a moderate correlation between expert feature-similarity and data-driven feature-similarity (Pearson's $\rho = 0.477$; Spearman's $r_s = 0.459$). Of the pairs, 1,005 had an expert feature-similarity of 1 (in the same state); however, only 110 (10.9%) of these had a data-driven feature-similarity of 1. This suggests that while our data-driven features measure assignment progress in a related way to the expert features, they may not agree exactly on whether two *different* students are in the same state. For comparison, we also calculated the correlation between the expert feature-similarity and the tree edit distance between the snapshots, and we found a much smaller, negative correlation (Pearson's $\rho = -0.002$; Spearman's $r_s = -0.130$). This suggests that there is essentially *no* linear relationship between tree edit distance and our expert feature-distance, and only a weak rank-order relationship. This is important because it means that our feature-state space picks up on very different properties of a student's code than just syntactic structure, making it a novel state representation. Additionally, it raises questions

about the utility of AST tree edit distance for comparing code snapshots, which is commonly used in data-driven program analysis, e.g. hint generation [8, 9].

3.2 Case Study

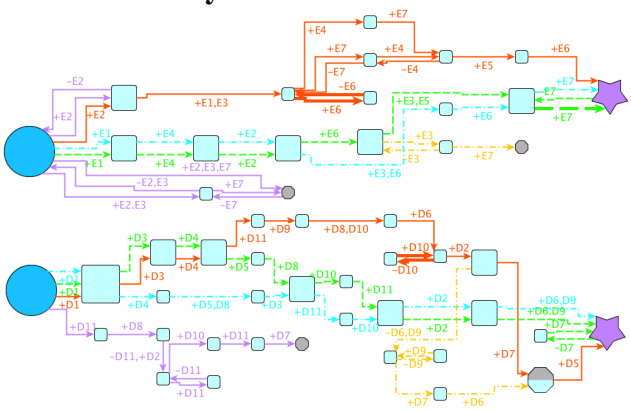

Figure 3: Select Student Progress through Expert (Top) and Data-Driven (Bottom) Feature-Spaces

Figure 3 shows an example of how 5 representative students progress through the feature-state space defined by both the expert-authored features (top) and the data-driven features (bottom). We manually selected these students (three of which achieved a correct solution) to showcase different problem-solving approaches. The graphs highlight similarities and differences between the two state spaces, and we present a few examples with additional details from the trace data. In both state spaces, the green and blue traces follow similar solution paths, often passing through the same state. However, the data-driven features show more divergence in the beginning because the green trace completed D3 (have a 'pen down' in a procedure) earlier, and this features has no expert analogue. In the red trace, both representations show that the student is struggling with which variable to use inside of the 'move' block, as the trace repeatedly gains and loses the analogous E6/D10 feature. However, in the expert state space, this struggling also affects the E4 and E7 features, which also involve those variables, but no other features are gained or lost in the data-driven state space. This shows how data-driven features can be less interrelated than the expert features, which sometimes share requirements or have strict dependencies. The expert state space also highlights more subtle differences in state, such as when the purple trace uses a variable that is set to 0 in its 'move' block. The expert state space shows the student losing feature E2 (draw anything), since the sprite does not actually move, but there is no equivalent change in the data-driven state space. For the yellow trace, our data-driven state space shows it overlapping multiple times with the red trace, while in the expert state space these traces are instead often in states that differ by only a single feature (e.g. E6: move variably). These examples support the conclusion from Section 3.1 that expert and data-driven state spaces share similar representations of student progress but still differ in meaningful ways.

4. CONCLUSION

This study proposed a novel way to reduce the state space of open-ended programming problems by defining a student's state using a set of features of a correct solution, giving clear meaning to states and transitions. We present a process by which these features can be defined by experts, and we show that this process can be automated with a data-driven approach. Our data-driven features differ from expert features, but they are also interpretable, many with a mapping to the expert features. They also showed moderate agreement with the expert features on when students changed state and on the similarity between students' states. A limitation of this work, however, is that it analyzed only one exemplar assignment in a small CS0 class using a block-based programming language, so its generalizability must be verified.

These preliminary results demonstrate that our feature-state space could have clear applications in educational data mining. Figures 2 and 3 show how our feature representation lends itself to visualizing a state space, which can allow educators and researchers to identify common solution strategies. This information can be integrated into an ITS to update a student model based on which features a student has completed, or to tailor intelligent support to a student's current feature-state. For example, an ITS could show a data-driven worked example *step*, showing a student how to complete a currently incomplete feature. Using previous students' data, we could additionally identify which features make the most sense for them to complete next, similar to how an interaction network [2] is used to generate next-step data-driven hints. Such data-driven worked examples are effective in scaffolding student progress in ITSs for other domains [5], and our compact feature-state space makes generating worked example steps feasible in programming as well. These potential applications and promising initial results motivate our continued research into meaningful and effective state space reduction.

5. REFERENCES

[1] N. Augsten, M. Böhlen, and J. Gamper. Approximate matching of hierarchical data using pq-grams. *VLDB*, pages 301–312, 2005.

[2] M. Eagle, M. Johnson, and T. Barnes. Interaction networks: Generating high level hints based on network community clustering. *EDM*, 2012.

[3] D. Garcia, B. Harvey, and T. Barnes. The Beauty and Joy of Computing. *ACM Inroads*, 6(4):71–79, 2015.

[4] A. Hicks, B. Peddycord III, and T. Barnes. Building games to learn from their players: Generating hints in a serious game. In *ITS*, pages 312–317, 2014.

[5] B. Mostafavi, G. Zhou, C. Lynch, M. Chi, and T. Barnes. Data-driven worked examples improve retention and completion in a logic tutor. In *AIED*, pages 726–729, 2015.

[6] T. W. Price and T. Barnes. An exploration of data-driven hint generation in an open-ended programming problem. In *GEDM Workshop*, 2015.

[7] T. W. Price, Y. Dong, and D. Lipovac. iSnap: towards intelligent tutoring in novice programming environments. In *SIGCSE*, pages 483–488, 2017.

[8] K. Rivers and K. R. Koedinger. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *AIED*, 27(1):37–64, 2017.

[9] K. Zimmerman and C. R. Rupakheti. An Automated Framework for Recommending Program Elements to Novices. In *ASE*, 2015.