Designing the Whyline: A Debugging Interface for Asking Questions about Program Behavior

Amy J. Ko and Brad A. Myers
Human-Computer Interaction Institute
Carnegie Mellon University
Pittsburgh, PA 15213
ajko@cmu.edu, bam+@cs.cmu.edu

Abstract

Debugging is still among the most common and costly of programming activities. One reason is that current debugging tools do not directly support the inquisitive nature of the activity. *Interrogative Debugging* is a new debugging paradigm in which programmers can ask *why did* and even *why didn't* questions directly about their program's runtime failures. The *Whyline* is a prototype Interrogative Debugging interface for the Alice programming environment that visualizes answers in terms of runtime events directly relevant to a programmer's question. Comparisons of identical debugging scenarios from user tests with and without the Whyline showed that the Whyline reduced debugging time by nearly a factor of 8, and helped programmers complete 40% more tasks.

Categories and Subject Descriptors: D.2.5. [**Testing and Debugging**]: Debugging aids.

Keywords: Debugging, Alice, program slicing.

INTRODUCTION

Among all programming activities, debugging still remains the most common and most costly. A recent study by the NIST found that software engineers in the U.S. spend 70-80% of their time testing and debugging, with the average error taking 17.4 hours to find and fix. Software engineers blamed inadequate testing and debugging tools [19].

One reason for this might be that the feature sets of commercial debugging tools have changed little in the past 30 years [14, 17]: programmers' only tools for finding errors are still breakpoints, code-stepping, and print statements. There have been many attempts to design more useful debugging paradigms and tools, including automatic debugging [1], relative debugging [21], program slicing [20, 22], and visualizations [6], but few of these have been shown to be usable, let alone to reduce debugging time. We believe this is because debugging activity always begins with a question, and to use existing tools, programmers

must struggle to map strategies for answering their question to the tools' limited capabilities. We propose to remove this hurdle by allowing programmers to directly ask the questions they naturally want to ask.

As part of the Natural Programming project (www.cs.cmu.edu/~natprog), we have found that programmers tend to ask *why did* or *why didn't* something happen. Thus, in this paper we propose a new debugging paradigm called *Interrogative Debugging*, in which programmers can ask *why did* and *why didn't* questions directly about a program's behavior, and view answers in terms of directly relevant runtime data. We then describe the design of the *Whyline*, an Interrogative Debugging interface for the Alice programming system. By comparing six identical debugging scenarios from user tests with and without the Whyline, we found that the Whyline reduced debugging time by nearly a factor of 8, enabling programmers to complete 40% more tasks.

INTERROGATIVE DEBUGGING

Research describes debugging as an exploratory activity aimed at investigating a program's behavior, involving several distinct and interleaving activities [4, 7, 15]:

- *Hypothesizing* what runtime actions caused failure;
- *Observing* data about a program's runtime state;
- *Restructuring* data into different representations;
- *Exploring* restructured runtime data;
- *Diagnosing* what code caused faulty runtime actions;
- *Repairing* erroneous code to prevent such actions.

Current debugging tools support some of these activities, while hindering others. For example, breakpoints and code-stepping support observation of control flow, but hinder exploration and restructuring [17]; visualization tools help restructure data, but hinder diagnosis and observation [11].

Yet none of these tools support *hypothesizing* activities. The argument behind *Interrogative Debugging* (ID) is that support for such question-related activities is essential to successful debugging. If programmers have a weak hypothesis about the cause of a failure, any implicit assumptions about what did or did not happen at runtime will go unchecked. Not only do these unchecked assumptions cause debugging to take more time [15], but we have shown that in Alice, 50% of all errors were due to programmers' false assumptions in the hypotheses they formed while debugging existing errors [12].

Thus, the fundamental goal of ID is to allow programmers to ask questions explicitly about their program's failure. For example, if a programmer was testing a graphical button and observed that clicking it had no effect, an ID interface would allow her to ask, "Why didn't this button do anything?" By allowing questions about program output, any implicit assumptions about what did or did not happen at runtime can be explicitly addressed in the answer.

In two studies of both experts' and novices' programming activity [12], programmers' questions at the time of failure were one of two types: *why did* questions, which assume the occurrence of an *unexpected* runtime action, and *why didn't* questions, which assume the absence of an *expected* runtime action. There were three possible answers:

1. *False propositions*. The programmer's assumption is false. The answer to "Why didn't this button's action happen?" may be that it did, but had no visible effect.

2. *Invariants*. The runtime action always happens (*why did*), or can never happen (*why didn't*). The answer to our button question may be that an event handler was not attached to an event, so it could never happen.

3. *Data and control flow*. A chain of runtime actions led to the program's output. For example, a conditional expression, which was supposed to fire the button's action, evaluated to false instead of true.

Interaction Design Requirements

Even with a clear idea of the functionality that an ID interface should have, there are numerous ways that one could design the interactions with such a tool. For example, some methods of *asking* questions may be better than others at allowing programmers to explore possible questions.

Some methods of *answering* questions might be better than others at revealing the important information. We propose a set of design constraints in Table 1, framed in terms of Green's Cognitive Dimensions Framework [9].

Consider *hidden dependencies*, which are entity relationships that are not fully visible. Gilmore argues that a critical part of debugging is searching for a set of runtime dependencies that explain a programmer's knowledge gap [7]. These important relationships should never be hidden, while the larger set of irrelevant dependencies could be hidden in trade for *visibility*. Commercial debugging tools are notorious for hidden dependencies: code stepping tools show runtime data on the call stack, but not the data that depends on it or that it depends on. Print statements reveal relevant runtime data, but hide the context from which the data was printed. Another issue is *viscosity*, or resistance to local changes. For example, inserting ten print statements to observe runtime events requires removing them later; trying to use a code stepping tool to show data from the past requires re-execution. These hindrances to exploration may lead to debugging errors.

THE WHYLINE

To investigate the merits of ID, we used our design constraints to design the Whyline—a **W**orkspace that **H**elps **Y**ou **L**ink **I**nstructions to **N**umbers and **E**vents. The Whyline is prototyped in Alice, the environment shown in Figure 1. Alice is an event-based language that simplifies the creation of interactive 3D worlds. Code is created by dragging and dropping tiles to the code area and choosing parameters from popup menus. This interaction prevents all type and syntax errors. See *www.alice.org* for more details.

Table 1. Proposed design constraints for debugging interfaces in terms of the Cognitive Dimensions Framework [8].

| Dimension	Rationale and Proposed Design Constraint
Error-proneness	Knowledge, attentional, and strategic breakdowns in debugging can cause errors, and should be prevented [12].
Premature Commitment	Assumptions and misperceptions about runtime history lead to inefficient, error-prone debugging strategies [5]. Don't require programmers to **hypothesize** and **diagnose** without **observations** of runtime data.
Viscosity	**Exploration** is essential to successful debugging [7], thus acquiring new runtime data should be unhindered.
Provisionality	There shouldn't be any commitment to the effects of exploring questions, runtime data, or code.
Visibility and Juxtaposability	Ability to see and compare code, runtime data and output is essential to successful **observation**, **hypothesizing** and **diagnosis** [7]. This type of data should have more prominence than other potentially visible information.
Hidden dependencies	With respect to a specific program failure, there are few relevant runtime dependencies and many irrelevant ones [7]. Emphasize relevant runtime dependencies by hiding irrelevant ones.
Secondary Notation	Support for meta-activities such as managing hypotheses and annotating possibly erroneous code may improve productivity [5]. Support secondary notations for annotating code, runtime data, and hypotheses considered.
Diffuseness	Verbosity of runtime actions should correlate with the programming language's verbosity.
Consistency	*Representations* of runtime actions should be understandable with only knowledge of the code.
Progressive Evaluation	Progressive evaluation is mentally simulating the execution of a program to test if a potential **repair** would prevent failure [14]. Debugging interfaces should help programmers simulate computation and control-flow.
Closeness of Mapping	**Diagnosis** involves understanding a computational machine's mapping from code to runtime behavior. Representation of runtime data should closely map to machine's behavior, in order to support diagnosis.
Role-Expressiveness	Programmers will not use a tool if they perceive no reward in using it [2]. Interface artifacts should have affordances for the actions they support and rewards they provide.
Hard Mental Operations	Understanding runtime causality (**hypothesizing** and **diagnosis**) are hard mental operations; other activities are not [7]. Make the easy activities easier and hard activities *possible*.
Abstraction-Gradient	Debugging is a tight **exploratory** cycle, suggesting that the reward for creating abstractions is small. Focus on helping programmers explore runtime data, not on abstracting runtime data.
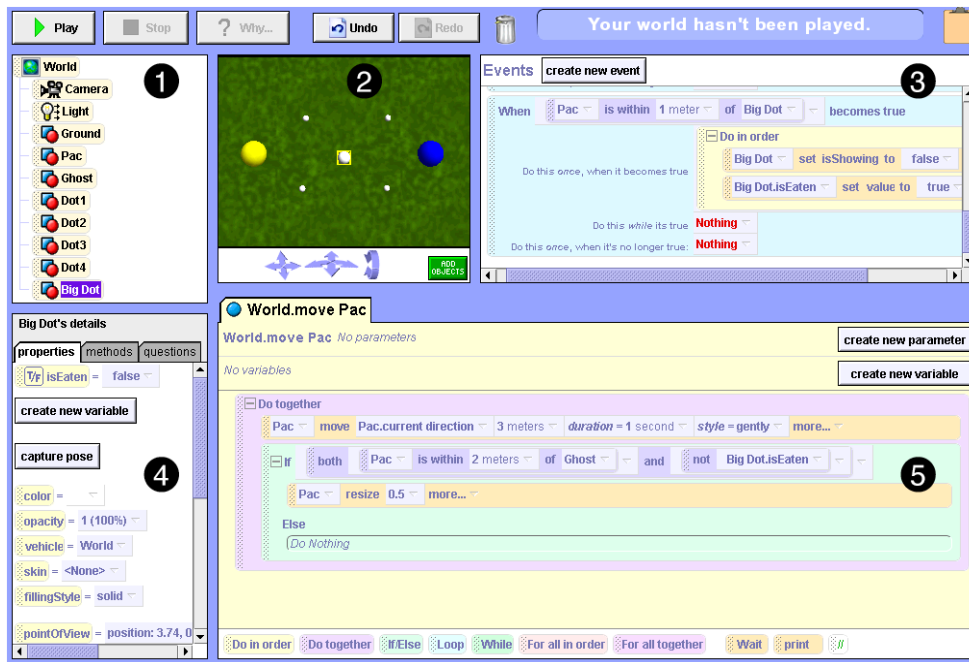
Figure 1. The Alice programming environment, before the world has been played: (1) the object list, (2) The 3D world view, (3) the event list, (4) the currently selected object's properties, methods, and questions, and (5) the code area.

Alice-Specific Constraints and Guidelines

To further inform our design, we performed two studies of novice and expert programmers using Alice in contextual and lab settings [12]. One major finding was that *50% of all errors were due to programmers' false assumptions in the hypotheses they formed while debugging existing errors*. There were also many consistent trends in their questions:

1. 68% of questions were *why didn't* questions and the remaining were *why did* questions.

2. Programmers only asked *why didn't* questions about code that plausibly might have executed.

3. 85% of questions were about a single object. The remaining concerned multiple objects' interactions.

An Illustration of the Whyline's Design

Before we describe the Whyline's implementation, let us illustrate its design through this debugging scenario (which comes directly from the user study we will discuss shortly):

Ellen is creating a Pac-Man game, and trying to make Pac shrink when the ghost is chasing and touches Pac.
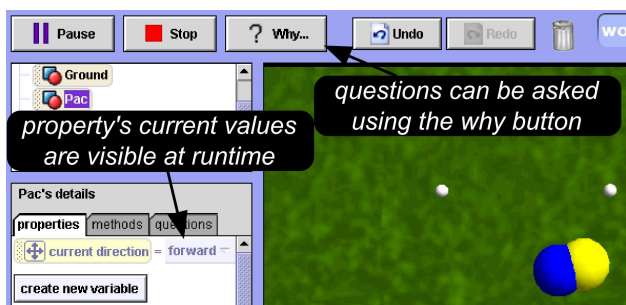

Figure 2. Ellen expected Pac to resize, but he did not.

She plays the world and makes Pac collide with the ghost, but to her surprise, Pac does not shrink...

Pac did not shrink because Ellen (a pseudonym) has code that prevents Pac from resizing after the big dot is eaten. Either Ellen did not notice that Pac ate the big dot, or she forgot about the dependency.

The Question Menu

When Ellen played the world, Alice hid the code and expanded the worldview and property panel, as seen in Figure 2. This relates property values to program output. Ellen presses the why button after noticing that Pac did not shrink, and a menu appears with the items *why did* and *why didn't*, as in Figure 3. The submenus contain the objects in the world that *were* or *could have been* affected. The menu
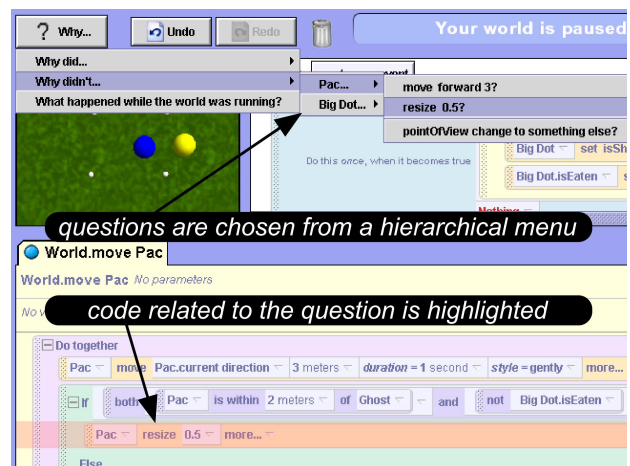

Figure 3. Ellen explores the questions and decides to ask "Why didn't Pac resize .5?" which highlights the code.

supports exploration and diagnosis by increasing questions' visibility and decreasing the viscosity of considering them.

Because Ellen expected Pac to resize after touching the ghost, she selects *why didn't Pac…* and scans the property changes and animations that could have happened. When she hovers the mouse over a menu item, the code that caused the output in question is highlighted and centered in the code area (see Figure 3). This supports diagnosis by exposing hidden dependencies between the failure and the code that might be responsible for it. This also avoids premature commitment in diagnosis by showing the subject of the question without requiring that the question be asked.

The Answer

Ellen asks *why didn't Pac resize .5?* and the camera focuses on Pac to increase his visibility. The Whyline answers the question by analyzing the runtime actions that did and did not happen, and provides the answer shown in Figure 4. The actions included are only those that prevented Pac from resizing: the predicate whose expression was false and the actions that defined the properties used by the expression. By excluding unrelated actions, we support observation and hypothesizing by increasing the visibility of the actions that likely contain the fault. To support diagnosis, the actions' names and colors are the same as the code that caused them.

This improves consistency and closeness of mapping with code.

The arrows represent data and control flow causality. Predicate arrows are labeled *true* or *false* and dataflow arrows are labeled with the data used by the action they point to. The arrows support progressive evaluation, and thus hypothesizing, by helping Ellen follow the runtime system's computation and control flow.

Along the x-axis is event-relative time, improving the closeness of mapping to the time-based Alice runtime system. Along the y-axis are event threads: this allows co-occurring events to be shown, supporting juxtaposibility.

Ellen interacts with the timeline by dragging the time cursor (the vertical black line in Figure 4). Doing so changes all properties to their values at the time represented by the time cursor's location. This supports exploration of runtime data. When Ellen moves the cursor over an action, the action and the code that caused it become selected, supporting diagnosis and repair. These features allow Ellen to rewind, fast-forward, and even "scrub" the execution history, receiving immediate feedback about the state of the world. This exposes hidden dependencies between actions and data that might not be shown directly on the Whyline, and between properties' current values and program output.
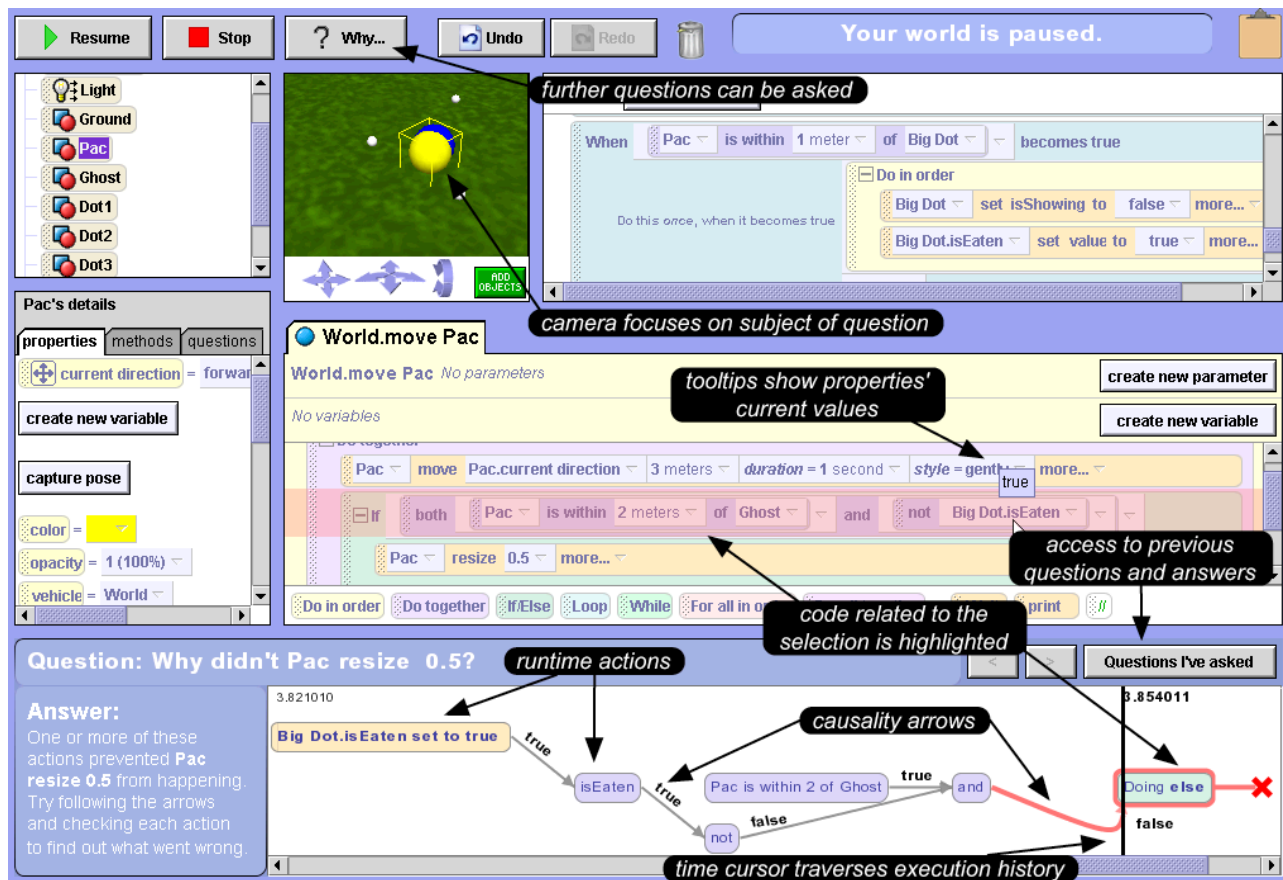

Figure 4. The Whyline's answer shows a visualization of the runtime actions preventing Pac from resizing. Ellen uses the time cursor to "scrub" the execution history, and realizes that Pac did not resize because *isEaten* was true.

To reduce the viscosity of exploration, Ellen can double-click on an action to implicitly ask *what caused this to happen?* and actions causing the runtime action are revealed. Ellen can also hover her mouse cursor over expressions in the code to see properties' current values and to evaluate expressions based on the current time. This improves the visibility of runtime data and supports progressive evaluation. Finally, the Whyline supports provisionality by making previous answers available through the *Questions I've Asked* button. The button prevents the hard mental operation of recalling facts determined earlier in debugging activity.

Ellen discovers her misperception using the Whyline:

> *"So this says Pac didn't resize because BigDot.isEaten is true…Oh! The ghost wasn't chasing because Pac ate the big dot. Let's try again without getting the big dot."*

Without the Whyline, the misperception could have led to an unnecessary search for non-existent errors.

Implementation

Alice programs are implemented internally using a control flow graph (CFG), where expressions are represented as data flow graphs attached to CFG nodes. These graphs are constructed incrementally as programmers create and modify code. At runtime, we annotate the graphs with all assignments and uses of properties' values. This value history is used to traverse the execution history based on the location of the time cursor.

We use static [20] and dynamic [22] analysis to support:

1. Creation of a database of all changes to output (all animation statements and changes to visible properties), used for generating the *why did* menu structure, and for answering false propositions.

2. Generation of precise dynamic slices on the code in question, used to visualize the Whyline's answer.

3. Generation of all *potential* definitions and uses of properties from static DFGs, used for constructing the *why didn't* question menu. We do not support continuous number spaces (i.e., an object's position), but do support the generic question "why didn't this property change?"

4. Answering of invariant *why didn't* questions by coupling a dynamic slice on an execution instance of a CFG node with the *reachability* of the node.

We used Alice-specific design heuristics to construct program slices. For example, for each question we had to choose which execution of the queried statement to slice on. Because our observations showed that questions were asked immediately after failures, we sliced on the most recent execution of a program statement.

For *why did* answers, we included up to two predicates in the queried runtime action's chain of causality, and any events defining the properties used in the predicates' expressions. Including more runtime actions would have

decreased visibility, and we believed forcing programmers to interactively reveal the history would better support progressive evaluation of their hypotheses.

For *why didn't* answers, we included all of the different executions of the predicate that prevented the code in question from being reached, and the *why did* slice on each.

USER STUDY

The Whyline was an exploration of an example Interrogative Debugging interface, so we had general questions about its usability and impact:

- Would the Whyline be considered useful?
- Would the Whyline reduce debugging time?
- Would the Whyline help complete more tasks?

To investigate these questions, we replicated the observational study discussed earlier [12], but included the Whyline in the Alice environment. We will refer to the first study without the Whyline as the *Without* study, and the present study as the *With* study. The *With* study used an iterative design methodology: observations from user session were used to fix usability problems and inform the design of features for successive sessions.

Participants and Method

In both studies, participants were recruited from the Carnegie Mellon HCI Masters program. Programming experience ranged from beginning Visual Basic to extensive C++ and Java. The four participants in the *Without* study will be referred to as C1-C4, and the five in the *With* study as E1-E5.

Sessions began with a 15-minute tutorial on creating Alice code. Participants were given the layout in Figure 1 and 90 minutes to make a Pac-Man game with these specifications:

1. Pac must always move. His direction should change in response to the arrow keys.

2. Ghost must move in random directions half of the time and directly towards Pac the other half.

3. If Ghost is chasing and touches Pac, Pac must flatten and stop moving forever.

4. If Pac eats the big dot, ghost must run away for 5 seconds, then return to chasing.

5. If Pac touches running ghost, Ghost must flatten and stop for 5 seconds, then chase again.

6. If Pac eats all of the dots, Ghost must stop and Pac must hop indefinitely.

For the purposes of analysis, we consider these six specifications to be six distinct tasks, since the code necessary to accomplish them were only related by the character's state of behavior.

In both studies, the experimenter used think-aloud and Contextual Inquiry [10] methods to track participants' goals, strategies and intents. Participants were also videotaped while they worked, for later analysis.

Table 2. Frequency of question/answer types in each study and times the Whyline was found useful for each.

Question Type	Answer Type	Frequency Question/Answer Pair was Asked		# of times Whyline was found useful in the *With* study
		Without (4 programmers)	*With* (5 programmers)	
Why Did	Invariant	0	0	0
	False Proposition	1	0	0
	Control/Data Flow	7	5	3
Total # of Why Did Questions		**8**	**5**	**3**
Why Didn't	Invariant	5	5	4
	False Proposition	5	7	5
	Control/Data Flow	7	7	7
Total # of Why Didn't Questions		**17**	**19**	**16**

Results

Table 2 shows the distributions of question/answer types in each study. *Why didn't* questions were more common than *why did* questions, and programmers rarely asked invariant or false proposition *why did* questions. Participants found the Whyline useful for 19 of 24 of their questions, and the proportion increased in successive user tests: the Whyline was useful for 0 of E1's 3 questions, but all 5 of E5's.

Table 3 describes six identical debugging scenarios that occurred in the *With* and *Without* studies (many other scenarios were similar, but not enough to warrant comparison). In the *Without* study, participants tended to hypothesize and diagnose by inspecting and rewriting code. In the *With* study, they tended to hypothesize and diagnose by asking questions and analyzing the Whyline's answer. A repeated measures ANOVA shows that in the six scenarios, the Whyline significantly decreased debugging time ($F_{1,5} = 12.64$, $p < .02$) by an average factor of 7.8.

Overall, in the 90 minutes allotted, programmers with the Whyline completed significantly more tasks (M = 3.20, SD = .457) than those without (M = 2.25, SD = .500), $t(7) = 3.0$, $p < .02$. This was a 40% increase in tasks completed.

DISCUSSION

The Whyline appears to have great potential as a usable and effective debugging tool for Alice users. In generalizing our experiences, there are many issues to consider.

Issues Raised in User Testing

In our user testing, there were a few significant usability issues with our initial designs, some of which we predicted, and others we did not. These have implications for the design of future ID interfaces.

In session 1, for example, our prototype did not support *why didn't* questions. When E1 first used the Whyline, he wanted to ask a *why didn't* question, but could not, and immediately decided "*this thing is useless.*" This suggests that support for *why didn't* questions may be crucial to programmers' perceptions of the utility of the tool.

In session 2, our prototype distinguished between questions about output statements (*why did*) and questions about property changes (which we phrased as *why is*). E2 observed a failure based on Pac's *direction* property and searched the *why did* menu, ignoring the *why is* menu. We asked her later if she had noticed the menu: "*I wanted to ask about something that already happened.*" This is

Table 3. Identical debugging scenarios in the With and Without studies. Scenarios are described by the programming error, the failure caused, each programmer's strategy and outcomes, and the time from failure to error diagnosis.

Programming Error and the Failure it Caused	Strategy, Outcome, and Time in Seconds from Failure → Error Diagnosis			
	Without		*With*	
Code resized Pac to 0, which Alice ignores → Pac doesn't resize after touching ghost.	Read events, moved method call to event, moved camera, toggled state variables. "*So it's the resize function that's not working.*"	**330** sec	Asked, *why didn't pac resize 0?* and got invariant answer. "*So resize to 0 must not work. I'll try .5 instead.*"	**38** sec
Forgot event to call method that moves ghost → ghost doesn't move after playing.	Stared at screen and held head in hands. "*Oh! I need an event to start it.*"	**75** sec	Browsed *why didn't ghost* questions and saw highlighted ghost movement code: "*...oh, I didn't call it anywhere!*"	**8** sec
Thought *dot2* was *dot1* and referenced wrong dot → *dot1* not eaten after collision.	Looked at code; searched for dot in world-view; removed then recreated collision code, this time without error.	**91** sec	Inspected *why didn't menu* and realized her misunderstanding: "*Oh, no questions about the other dots. That must be dot2*".	**9** sec
Maps right key to *right* direction → Pac moves down instead of right.	"*I think this is wrong; it doesn't go to where I want it to be...this is terrible!*" Tried all possible mappings until she realized direc-	**182** sec	Asked *why did Pac move right?* Inspected control and data flow answer; noticed direction set to right because down was pressed: "*So direction is relative*	**28** sec
Dot collision threshold too small for test to happen while Pac is over dot → dot not	"*I made some methods that I thought would help me rid of the dots...I'm pretty sure I got close enough.*" Rewrote collision events and	**207** sec	Asked *why didn't dot1.isShowing change to false?* Scrubbed predicate tests: "*that's really intuitive...so when it actually did this test, this was the state of*	**27** sec
Forgot event to call *eatBigDot* method → Dot not eaten after touching Pac	"*There is definitely nothing happening.*" Browsed and inspected code and event list. "*Oh, of course not!*"	**49** sec	Asked *why didn't big dot isShowing change to false?* and read invariant answer which noted that nothing called the method. "*Oh, you're right!*"	**10** sec

consistent with our observations that programmers phrased questions in terms of failures instead of runtime data: she said *why did Pac's direction change to forward*? and not *why is Pac's direction forward right now?*

In session 3, our prototype answered questions relative to the time cursor's placement. When E3 asked his first question, he moved the time cursor, and upon asking his second question, noticed that the contents of the question menu changed considerably: "*Where did all my questions go?*" This was our rationale for using a statement's latest execution, regardless of the time cursor's placement.

Usability issues were also found in sessions 4 and 5, but not of the same magnitude as in the first three.

What was Helpful?

The most helpful feature of the Whyline seemed to be the question menu. Observations confirmed our hypothesis that asking questions in terms of program output, rather than code or runtime actions, would make it easier for programmers to map their question to related code. By restricting the programmer's ability to make assumptions about what did and did not happen, we enabled them to observe and explore the runtime actions that most likely caused failures. We plan to further refine the Whyline and formally investigate these findings.

Similarly, relating code to runtime actions interactively with the time cursor and visual highlighting helped with diagnosis and repair activities, as predicted. Had this relationship not been explicitly visualized, more text would have been needed to denote what caused the runtime action, decreasing visibility, and programmers would have had to manually search for the code responsible.

Finally, the data and control flow arrows directly supported hypothesizing which runtime events caused failure, as predicted. This seemed to be because the visualization acted as an external memory aid to help programmers simulate runtime execution. In the *Without* study, participants were forced to calculate expressions manually, allowing for attentional breakdowns during calculation.

When the time cursor, reversibility, and other features were used, our observations suggest that they played supporting roles in the Whyline's overall effectiveness.

FUTURE WORK

Making the Whyline More Helpful

Currently, questions about complex Boolean and numerical expressions give equally complex answers. This is because the level of detail in the questions is not enough to know which particular part of the data flow path is problematic. Reichwein et al. describe one solution that allows spreadsheet users to mark intermediate values in dataflow paths as correct or incorrect, which feeds into a visualization of which computations may be faulty [16].

Programmers often needed to inspect the internals of Alice primitives. For example, choosing the *distance* for "is object *a* within *distance* of object *b*" was difficult, because

programmers could not see the values used by the internal inequality at runtime. One solution would be to instrument the surface-level internal logic of primitives, so that such expressions could be shown on the Whyline.

The Whyline does not yet support *object-relative* questions (such as "why did Pac resize *after* Ghost moved"), which were fairly common in early observations of Alice programmers. In future prototypes, we will investigate constructing object-relative questions using direct manipulation of the objects on the screen.

In our user studies, using the latest execution of the queried statement was sufficient. In more complicated Alice worlds, this may not hold true. One possible interaction would allow programmers to further specify their questions with a time, which would allow them to find a particular execution in the recent history. This may require domain-specific heuristics to design effectively.

We plan to make the Whyline available online, and deploy it to the several universities where Alice is already in use.

Interrogative Debugging for Other Languages

"Gosh, that's really intuitive. Can you make this for Java?"
Subject E5, after his first program failure

There are many open questions in designing ID tools for other languages. For example, *for a given language and programming task, what output will programmers want to ask about*? In a modern code base, output might be numerical, message-based, or simply the execution of a stub of code. One way to automatically determine the granularity and nature of such output would be to search for previous executions of code in recently modified source files. This analysis could be used to select a set of output statements for *why did* and *why didn't* questions that are likely to be relevant. Even still, the sheer number of possible questions would be overwhelming. Future work must devise methods for selecting a small set of contextually relevant questions from all possible questions.

Because our implementation requires the complete execution history, another issue is *memory and performance*. Researchers have developed time- and space-efficient approaches to recording data definitions and uses, building CFGs and DFGs, and generating dynamic slices [20, 22]. However, how to efficiently generate a set of relevant questions remains an open question. Even with these methods, failures that are difficult to reproduce or take significant time to reproduce will pose significant challenges.

Another question is, *for a given task and language, what heuristics generate the most understandable answers?* We only included a small portion of a dynamic slice because of the simplicity of most Alice worlds. For more complex software, there would be a host of visualization and interactive issues in presenting a dynamic slice. One approach might be to use *semantic differencing* methods [18] to determine what parts of a dynamic slice might be anomalous, based on past executions histories.

RELATED WORK

There are a few systems with approaches similar to ID. Lencevicius et al. discuss *Query-Based Debugging* [13], where programmers form textual queries on objects' runtime relationships. However, it forces programmers to guess what relationships *might* exist, and requires learning an unfamiliar query language. Briggs et al. discuss a *task timeline* [3] for debugging distributed Ada programs. Their visualization is similar to the Whyline's in that it highlights a dynamic slice, but it does not relate runtime events to code. Zeller's work on cause-effect chains and the AskIgor debugger [21] is a related diagnosis tool. However, Zeller's approach requires both a failed and successful execution of a program. ID interfaces have no such requirement.

CONCLUSIONS

We have discussed a new debugging paradigm, called Interrogative Debugging, and a new debugging interface called the Whyline. The Whyline allows programmers to ask *why did* and *why didn't* questions about runtime failures, and provides direct access to the runtime data that they need to debug. Our user studies demonstrate that the Whyline can dramatically reduce debugging time, suggesting the potential of Interrogative Debugging as a highly effective approach to supporting debugging activity.

ACKNOWLEDGEMENTS

REFERENCES

[1] Auguston, M., Jeffery, C., and Underwood, S. A framework for automatic debugging, *IEEE International Conference on Automated Software Engineering,* 2002, Edinburgh, UK, 217-222.

[2] Blackwell, A. and Burnett, M. Applying attention investment to end-user programming, *IEEE Symposia on Human-Centric Computing Languages and Environments,* 2002, Arlington, VA, 28-30.

[3] Briggs, J.S*., et al.*, Task time lines as a debugging tool. *ACM SIGAda Ada Letters*, XVI(2), 1996, 50-69.

[4] Davies, S.P., Models and theories of programming strategy. *International Journal of Man-Machine Studies*, (39), 1993, 236-267.

[5] Davies, S.P. Display-based problem solving strategies in computer programming, *Empirical Studies of Programmers, 6th Workshop,* 1996, Washington, D.C., 59-76.

[6] Gestwicki, P. and Jayaraman, B. Interactive visualization of java programs, *IEEE Symposia on Human-Centric Computing Languages and Environments,* 2002, Arlington, VA, 226-235.

[7] Gilmore, D.J., Models of debugging. *Acta Psychologica*, (78), 1992, 151-173.

[8] Green, T.R.G., Cognitive dimensions of notations, in *People and computers v*, A. Sutcliffe and L. Macaulay, Editors. 1989, Cambridge, UK, Cambridge University Press, 443-460.

[9] Green, T.R.G. and Petre, M., Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7, 1996, 131-174.

[10] Holtzblatt, K. and Beyer, H., *Contextual design: Defining customer-centered systems*. 1998, San Francisco, CA, Morgan Kaufmann.

[11] Kehoe, C., Stasko, J., and Taylor, A., Rethinking the evaluation of algorithm animations as learning aids: An observational study. *International Journal of Human-Computer Studies*, 54(2), 2001, 265-284.

[12] Ko, A.J. and Myers, B.A. Development and evaluation of a model of programming errors, *IEEE Symposia on Human-Centric Computing Languages and Environments,* 2003, Auckland, New Zealand, 7-14.

[13] Lencevicius, R., Holzle, U., and Singh, A.K., Dynamic query-based debugging of object-oriented programs. *Journal of Automated Software Engineering*, 10(1), 2003, 367-370.

[14] Lieberman, H., The debugging scandal and what to do about it. *Communications of the ACM*, 40(4), 1997, 26-78.

[15] Mayrhauser, A.v. and Vans, A.M. Program understanding behavior during debugging of large scale software, *Empirical Studies of Programmers, 7th Workshop,* 1997, Alexandria, VA, 157-179.

[16] Reichwein, J., Rothermel, G., and Burnett, M. Slicing spreadsheets: An integrated methodology for spreadsheet testing and debugging, *Proceedings of the 2nd Conference on Domain-specific Languages,* 2000, Austin, Texas, 25-38.

[17] Romero, P*., et al.*, A survey of external representations employed in object-oriented programming environments. *Journal of Visual Languages and Computing*, 14, 2003, 387-419.

[18] Rothermel, G., Harrold, M.J., and Dedhia, J., Regression test selection for c++ software. *Software Testing, Verification & Reliability*, 10(2), 2000, 77-109.

[19] Tassey, G., *The economic impacts of inadequate infrastructure for software testing*. National Institute of Standards and Technology, RTI Project Number 7007.011, 2002.

[20] Tip, F., A survey of programming slicing techniques. *Journal of Programming Languages*, 3, 1995, 121-189.

[21] Zeller, A. Isolating cause-effect chains from computer programs, *International Symposium on the Foundations of Software Engineering,* 2002, Charleston, SC, 1-10.

[22] Zhang, X. and Zhang, Y. Precise dynamic slicing algorithms, *International Conference on Software Engineering,* 2003, Portland, OR, 319-329.