Helping Users Avoid Bugs in GUI Applications

Amir Michail

School of Computer Science & Eng
Univ of New South Wales
Sydney, Australia

Tao Xie

Dept. of Computer Science & Eng
Univ of Washington
Seattle, USA

Introduction

- Nowadays, majority of productivity applications are interactive and graphical in nature

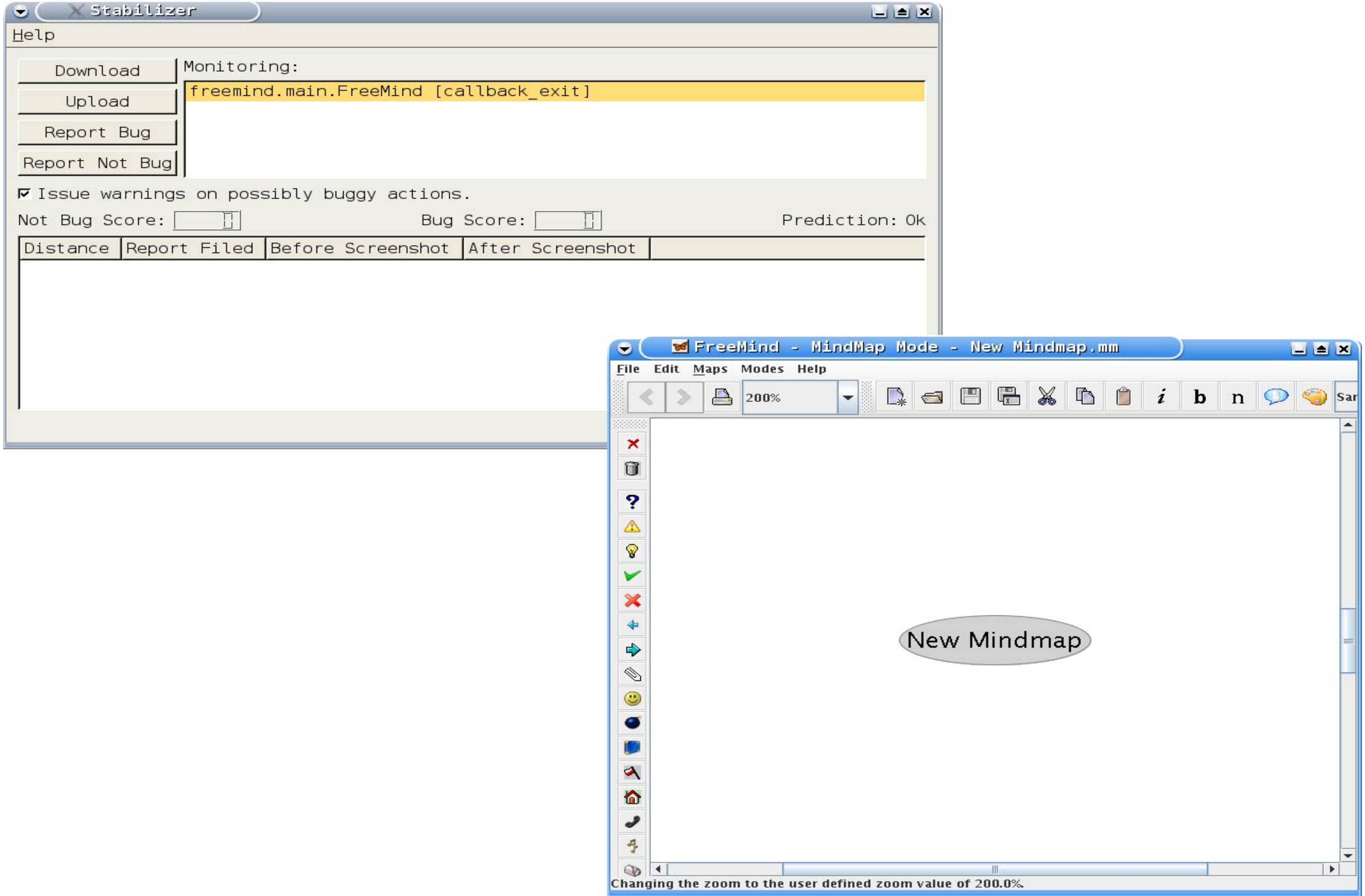
- (Both GUI and non-GUI) applications are buggy
  - bug number: Mozilla browser (20,000 open bugs)
  - bug life: Linux bugs (average 1.8 yrs, median 1.25 yrs)

- We take advantage of GUI-callback characteristics and machine learning in a new tool called Stabilizer
  - GUI callbacks can often be aborted without damaging app exec.

- Stabilizer helps users avoid bugs in GUI applications
  - allow users to collaboratively help each other avoid bugs
  - make a buggy application more usable in the meantime

Run FreeMind (Buggy App) with Stabilizer

(1) create a new mind map



Monitoring:

freemind.main.FreeMind [callback_exit]

☑ Issue warnings on possibly buggy actions.

Not Bug Score:

Bug Score:

Prediction: Ok

Distance	Report Filed	Before Screenshot	After Screenshot

200%

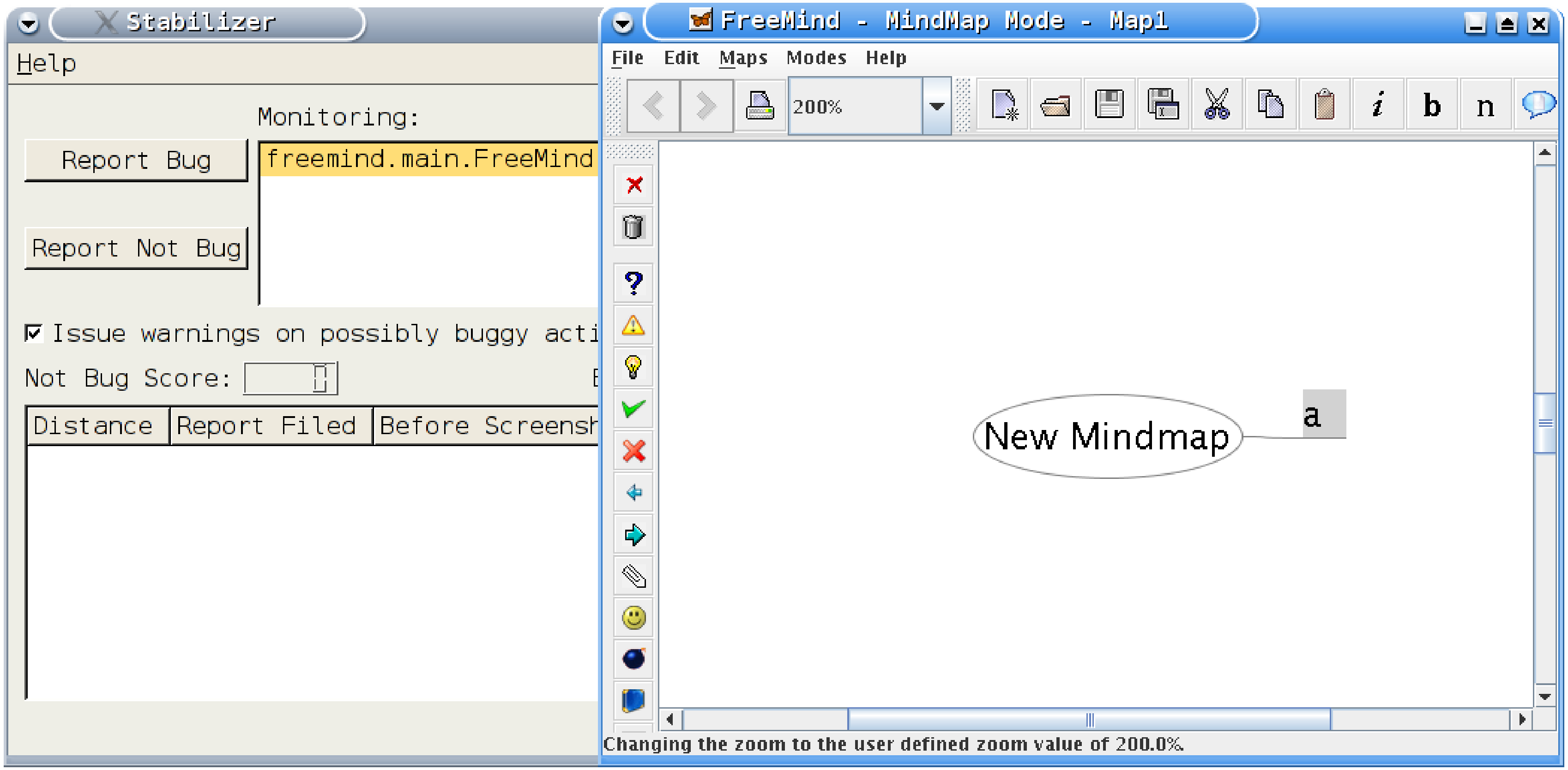
Changing the zoom to the user defined zoom value of 200.0%.

Add child node

(1) press *F10* to access the menu

(2) use the keyboard to select menu item *Edit ➔ New Child Node*

(3) type "a" as the text for the newly created child node.



Changing the zoom to the user defined zoom value of 200.0%.

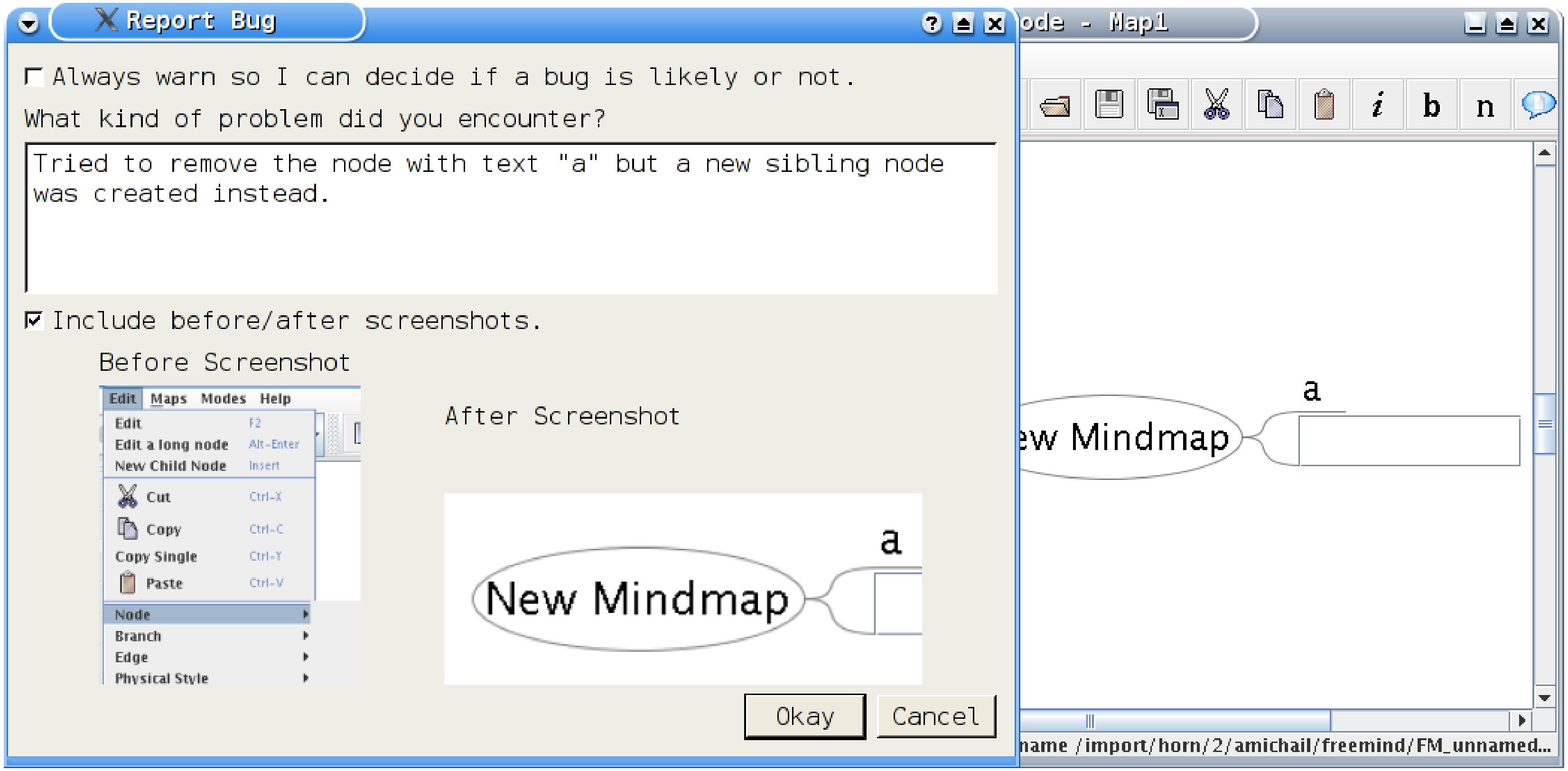
Delete child node

(1) press *F10* to access the menu

(2) use the keyboard to select the menu item *Edit* ➔ *Node* ➔ *Remove Node*

(3) observe a bug! the child node was not deleted, instead a sibling node was created. So now the root has two children.

(4) press *F11* (report-bug shortcut), a "Report Bug" dialog is popped up.



☐ Always warn so I can decide if a bug is likely or not.

What kind of problem did you encounter?

Tried to remove the node with text "a" but a new sibling node was created instead.

☑ Include before/after screenshots.

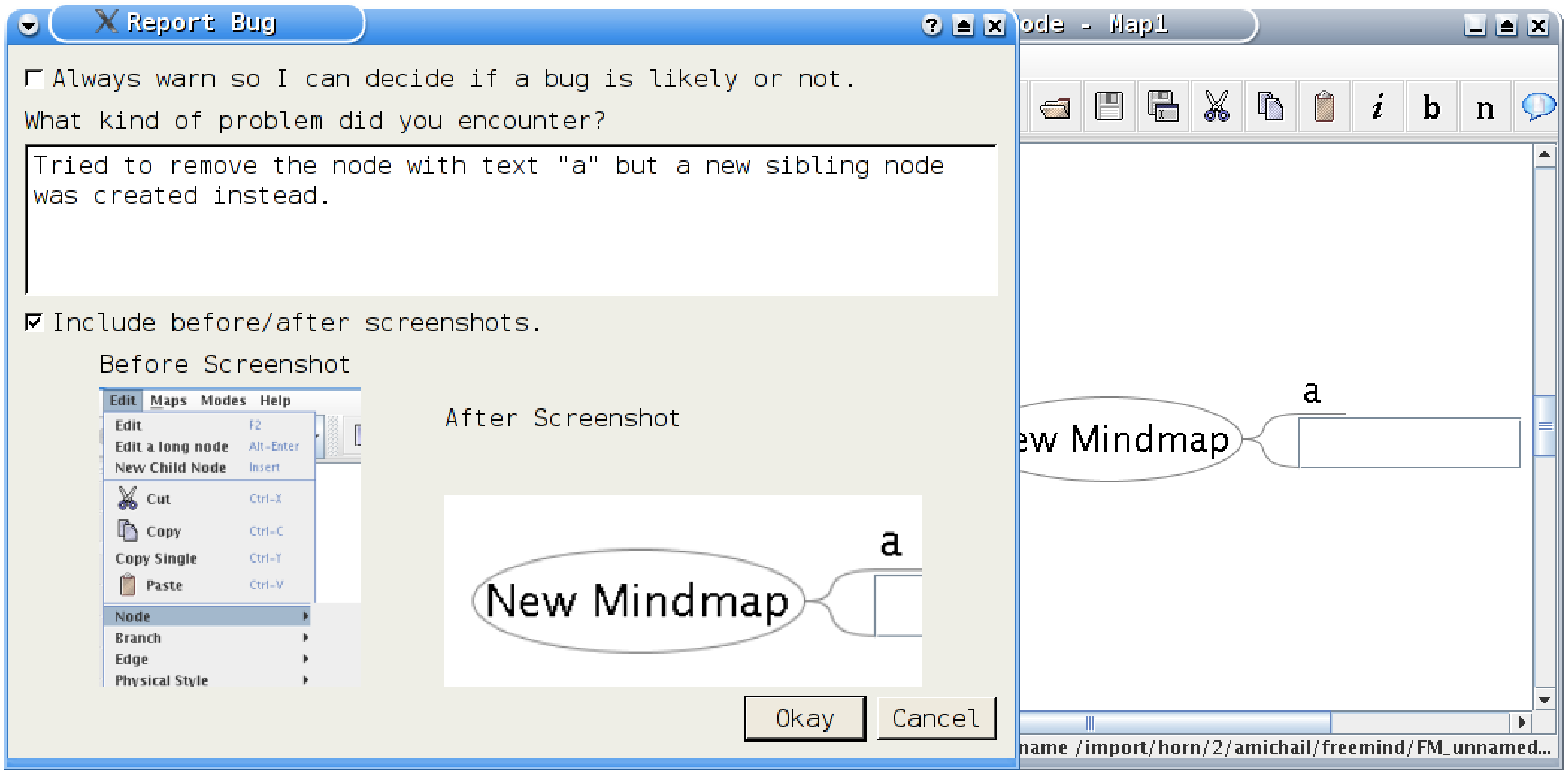
Before Screenshot

After Screenshot

Report bug

(1) in *text description,* explain what happened in words

(2) in *visual description,* use the mouse to zoom in on the relevant parts of the before and after screenshots
(although entire before/after screenshots are taken automatically)



☐ Always warn so I can decide if a bug is likely or not.

What kind of problem did you encounter?

Tried to remove the node with text "a" but a new sibling node was created instead.

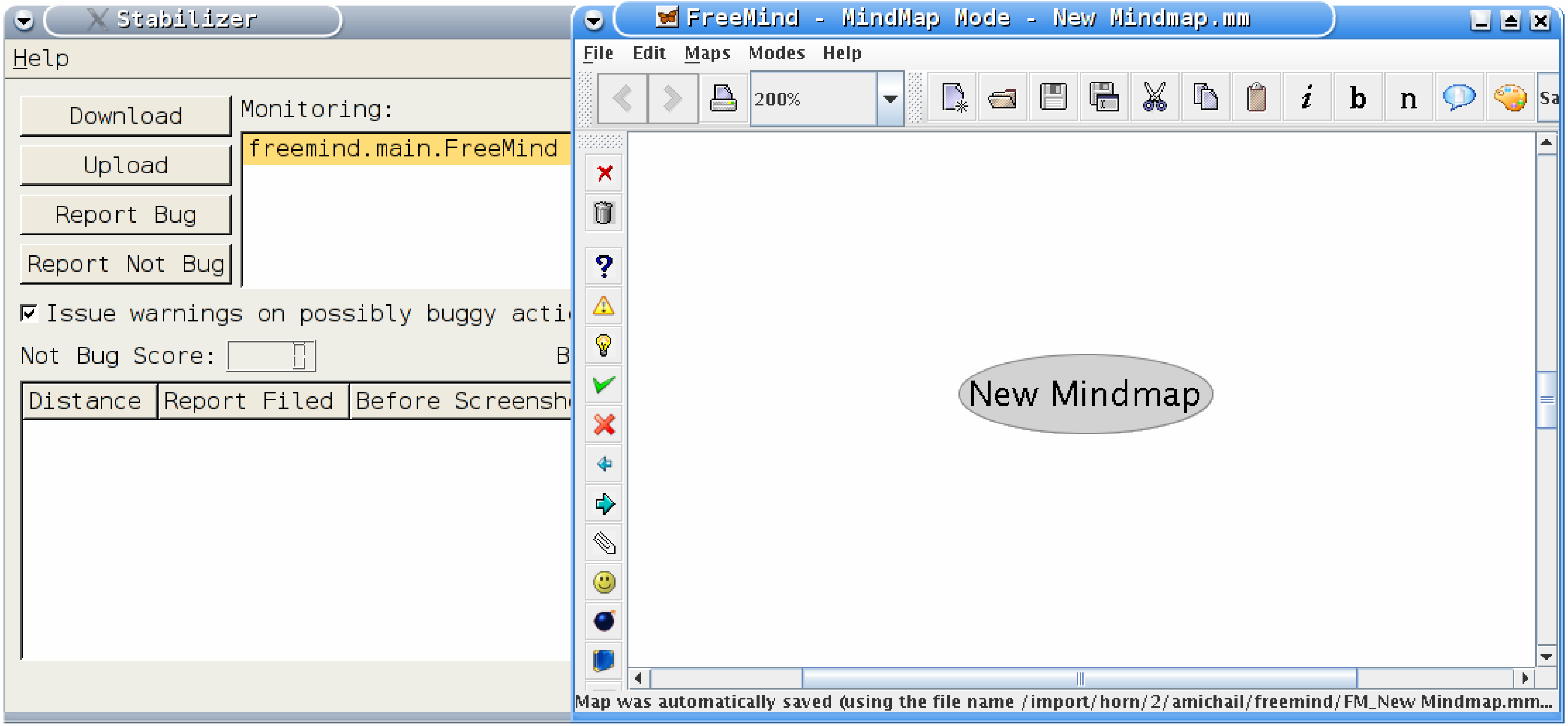
☑ Include before/after screenshots.

Delete child node differently

(1) click the right mouse button for the popup menu (rather than *F10* for the menubar)

(2) select the menu item *Node* ➔ *Remove Node*

(3) observe no bug! the child was indeed deleted as expected.



Monitoring: freemind.main.FreeMind

☑ Issue warnings on possibly buggy acti

Not Bug Score:

Map was automatically saved (using the file name /import/horn/2/amichail/freemind/FM_New Mindmap.mm...

Delete added child node again

…by the same user later or a different user

(1) add a child node whose text is “b” (following similar steps as before)
(2) press *F10* to access the menu
(3) use the keyboard to select the menu item *Edit ➔ Node ➔ Remove Node*
(4) get a warning — the same bug encountered before
(5) click *Abort Action* button to avoid the bug

Some user(s) reported a bug after an action similar to this one.
Would you like to abort the action?

Reports filed after an action similar to this one:

Distanc	Report Filed	Before Screenshot	After Screenshot
0.29...	Bug: Tried to remove the n...		
0.38...	Not Bug: No bug.		

Why not avoid bugs manually?

— Why need Stabilizer?

- Remembering bugs imposes heavy memory burden
  - an app may have many bugs
  - new releases may fix old bugs and introduce new bugs
  - many apps used by a user may have bugs
- Not easy for users to learn from other users
  - better if avoid a bug without even encountering it once.
  - but unrealistic to read and remember bug reports in Bugzilla
- Require to figure out the circumstances under which a bug occurs
  - not easy to identify the bug exposure conditions
  - made easier if pulling together execution context from many users

Now for the details …

Learn from the past to avoid buggy actions

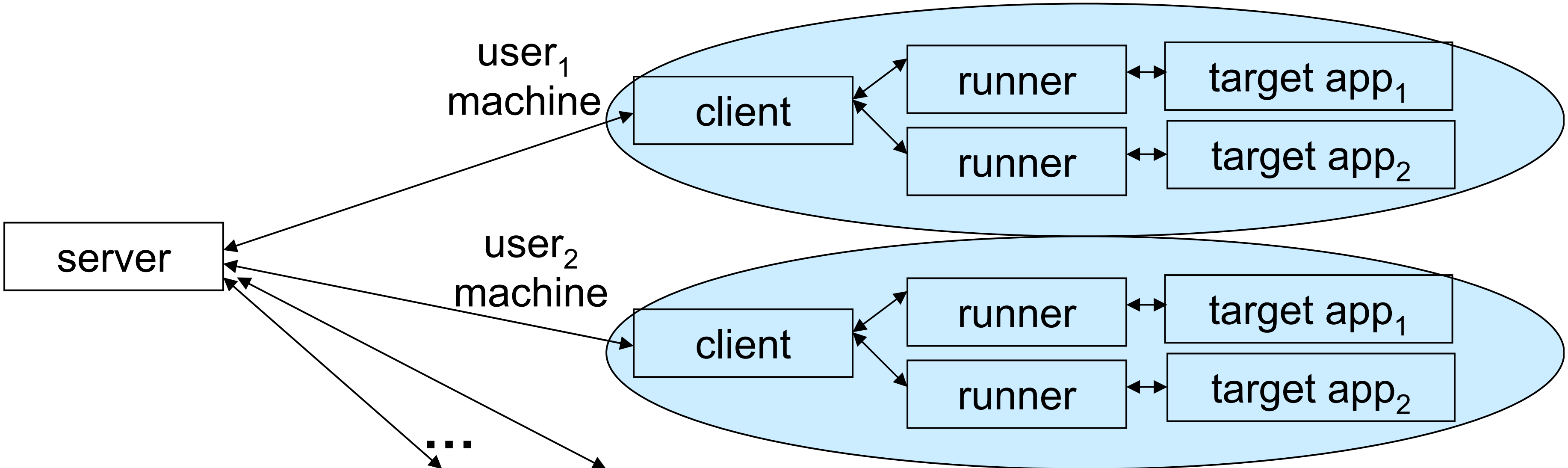
- How to define an **action**?
  - less useful to get a warning when bad things already happened or are unavoidable
  - good news: user action ⇔ event (callback) in GUI apps
  - challenge: action execution depends on context ➔ approximate context with bounded execution history

- How to know it was a **bad** or **good** past of an action?
  - crash or not; “bug” and “not bug” report

- How to **predict** based on learning from the past?
  - distance weighted nearest neighbor

Stabilizer architecture

- Stabilizer runner
  - run target app, collect runtime info, abort callbacks to avoid bugs
- Stabilizer server
  - central bug reporting server
- Stabilizer client
  - run on user's computer that monitors target app
  - make prediction
    - download historical samples from server at runner startup
    - upload new samples to server at runner shutdown



How to define an action?

- Action:
  - application state S (context) and an event e
- Approximate S with bounded exec history H
  - event history: H_e
  - code history: H_c (either function calls or basic blocks)

$$\begin{array}{l} H = (h_1, \ldots, h_n) \\ H = (\ldots, h_i, x, h_j, \ldots) \end{array} \quad + \quad \text{item: } x \quad \rightarrow \quad \begin{array}{l} H = (h_1, \ldots, h_n, x) \\ H = (\ldots, h_i, h_j, \ldots, x) \end{array}$$

How to know bad or good past

- Report “bug”
  - observe buggy behavior
  - press report-bug shortcut to report bug

  client adds a training sample: (H_e, H_c, “bug”)

- Report “not bug”
  - continue the action even when a bug warning is issued
  - observe bug-free behavior
  - press report-not-bug shortcut to report not bug

  client adds a training sample: ($H_e^{p,w}$, $H_c^{p,w}$, “not bug”) ,

  $H_e^{p,w}$ ends with the most recent event e_w

  $H_c^{p,w}$ contains the code history leading up to e_w

How to predict?

- Idea: consider the closest *k* training samples to see whether a bug is likely for some $k > 1$
- Given (H_e^p, H_c^p), for each sample $(H_e', H_c', type)$
  - measure distance : $0 \leq d((H_e^p, H_c^p), (H_e', H_c')) \leq 1$
  - if some $d == 0$, take type majority vote
  - otherwise, consider the closest k training examples, see which score is higher:
    - "bug" score

$$\sum_{(H_e', H_c', \text{``bug''}) \in X} \frac{1}{d((H_e^p, H_c^p), (H_e', H_c'))^2}$$

    - "not bug" score

$$\sum_{(H_e', H_c', \text{``not bug''}) \in Y} \frac{1}{d((H_e^p, H_c^p), (H_e', H_c'))^2}$$

Distance measure used in learner

- If the last event in H_e^p is not present in H_e', d ==1
- otherwise, compute the standard cosine similarity from info retrieval [Witten et al. 99]

event history
$$S_e(H_e^p, H_e') = \frac{\sum_{x \in H_e^p \cap H_e'} w_e^p(x) w_e'(x)}{\sqrt{\sum_{x \in H_e^p} w_e^p(x)^2} \sqrt{\sum_{x \in H_e'} w_e'(x)^2}}$$

code history
$$S_c(H_c^p, H_c') = \frac{\sum_{x \in H_c^p \cap H_c'} w_c^p(x) w_c'(x)}{\sqrt{\sum_{x \in H_c^p} w_c^p(x)^2} \sqrt{\sum_{x \in H_c'} w_c'(x)^2}}$$

combined similarity
$$S((H_e^p, H_c^p), (H_e', H_c') = \alpha S(H_e^p, H_e') + (1 - \alpha) S(H_c^p, H_c')$$

$$d((H_e^p, H_c^p), (H_e', H_c') = 1 - S((H_e^p, H_c^p), (H_e', H_c')$$

Evaluation of bug predication

Investigate three research questions

- can event history or code history be useful?
(i.e., regular method calls or basic blocks)

- can lower-level exec info be useful?
(i.e., arg of event callbacks or arg/ret of regular method calls)?

- can the Stabilizer's automated bug prediction be improved over learning time?

Experimental subjects [Memon et al. 03]

program	loc	classes	det mutants	indet mutants	tests
TerpWord	1747	9	17	2	170
TerpPresent	4769	4	5	9	56
TerpPaint	9287	42	8	0	—
TerpSheet	9964	25	3	6	152

- simulation of user interactions: run GUI tests
- bug exposure: manually write exposure conditions around the mutated lines
  - det mutants: whenever a callback is executed, bug is exposed (easy cases for Stabilizer)
  - indet mutants: otherwise (our evaluation focus)

Configurations of Stabilizer

Investigate effects of

event history, code history, lower-level exec info, learning time

Default Config (DC):

use only events and event callback arguments

event history size is 10

- Config 1: DC but no event callback arguments
- Config 2: DC
- Config 3: DC and method calls
- Config 4: DC and method calls with arg/ret values
- Config 5: DC and basic blocks
- Config 6: DC but event history size is 5
- Config 7: DC but event history size is 2
- Config 8: DC but event history size is 1

Measurements of bug prediction

- Compare bug predictions to actual bug occurrences
- Standard measures from info retrieval [Witten et al. 99]
  - Precision:

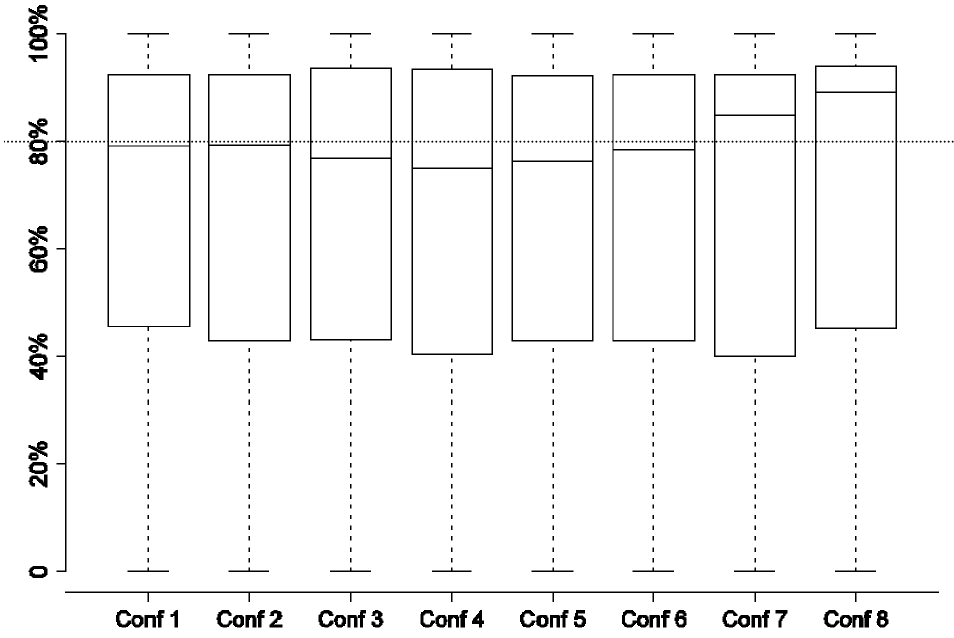
$$\frac{\text{\# correctly predicted buggy events}}{\text{\# bug warnings}}$$

  - Recall:

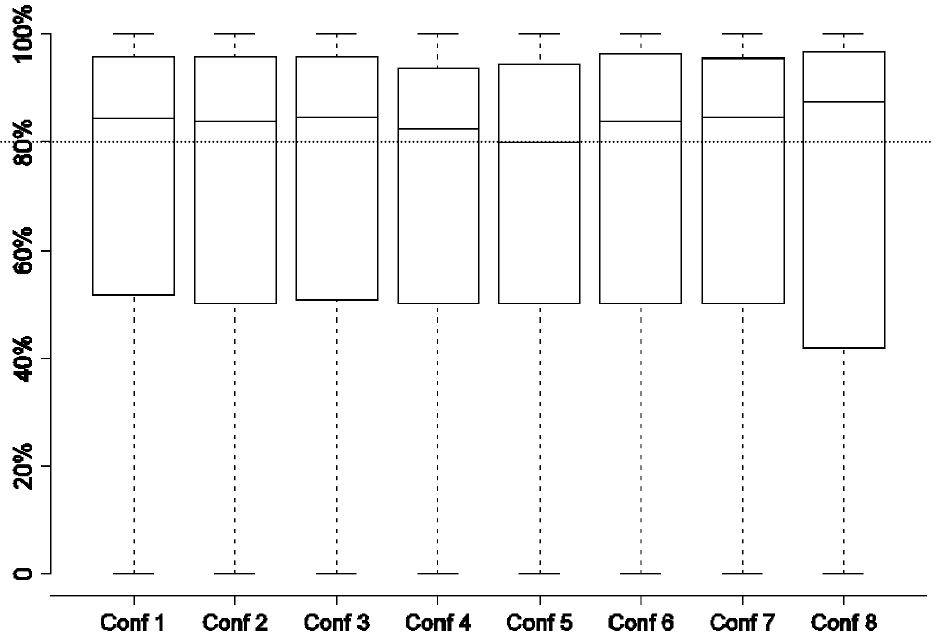
$$\frac{\text{\#correctly predicted buggy events}}{\text{\#events that were actually buggy}}$$

Experimental results - precision

$$\frac{\#\ \text{correctly predicted buggy events}}{\#\ \text{bug warnings}}$$



over the whole period

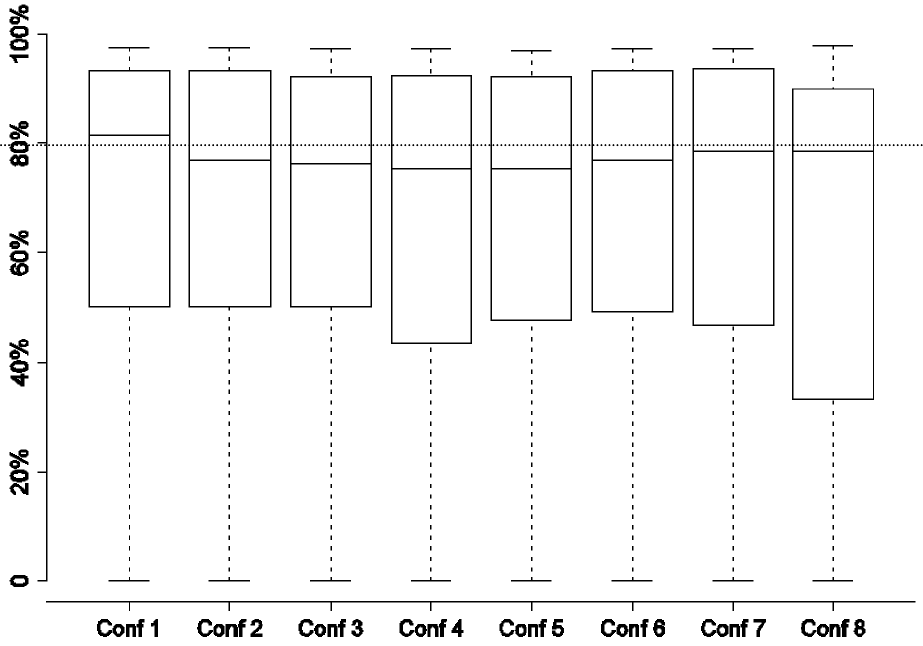


over the 2rd part of the period

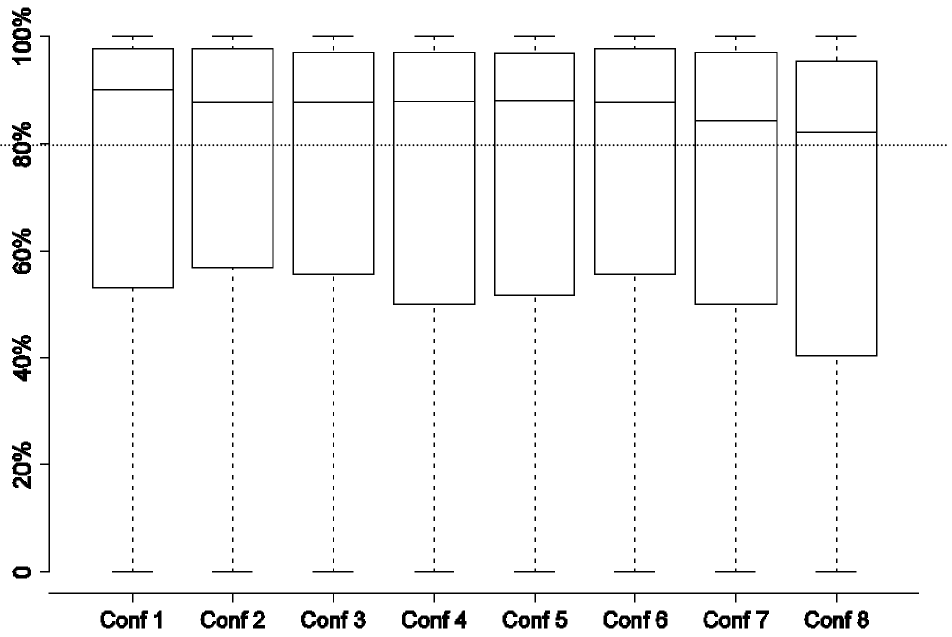
- ~80% median for precision
- event history, code history, or lower-level exec info does make a big difference
  but event history can be important in FreeMind case study
- improved over time (slightly)

Experimental results - recall

$$\frac{\#\text{correctly predicted buggy events}}{\#\text{events that were actually buggy}}$$



over the whole period



over the 2rd part of the period

- ~80% median for recall
- event history, code history, or lower-level exec info does make a big difference

  but event history can be important in FreeMind case study
- improved over time (more significantly)

Related work

- Cooperative bug isolation [Liblit et al. 03]
  - consider program crashes vs. undesirable behavior as a bug
  - help app developers vs. app users
  - use exec info available before crash site vs. before buggy call back
  - human-understandable bug conditions vs. not required
- Delta debugging [Zeller et al. 02]
  - proactively generate tests vs. exploit collective historical execution

Related work (cont.)

- Data structure repairing [Demsky & Rinard 03]
  - require specifications vs. not require
  - aggressively repair vs. avoid entering a corrupted state
- Anomalies as precursors of field failures [Elbaum et al. 03]
  - normal behaviors: in-house testing vs. callback's passing runs
  - abnormal behaviors: deviated in-field runs vs. callback's failing runs
- Intrusion detection with the sliding window nearest neighbor method [Lane&Brodley 99]

Conclusion

- A tool-based approach to help users avoid bugs in GUI apps.
- Users would use the app normally and report bugs (and also “not bugs") that they encounter
  - prevent anyone —including themselves— from encountering those bugs again
- Future work
  - improve bug prediction
    - look at app state info
    - look ahead by forking child processes
  - evaluation on many users, including non-technical ones
    - Stabilizer being developed with distributed operation in mind

http://cgi.cse.unsw.edu.au/~stabilizer/

Questions?

Problem statement

- Problem statement: given an application state *S* (context) and an event *e*, would processing event *e* in state *S* likely result in a bug given past bug and “not bug" reports?

- A bounded execution history to approximate *S*
  - event history
  - code history (either function calls or basic blocks)