



# *Warum stürzt mein Programm ab?*

## *Automatisches Isolieren von Fehlerursachen*

Andreas Zeller

Lehrstuhl Softwaretechnik  
Universität des Saarlandes, Saarbrücken



# Eine wahre Geschichte

---



Wir betrachten das folgende C-Programm:

```
double bug(double z[], int n) {  
    int i, j;  
    i = 0;  
    for (j = 0; j < n; j++) {  
        i = i + j + 1;  
        z[i] = z[i] * (z[0] + 1.0);  
    }  
    return z[n];  
}
```

bug.c läßt den GNU-Compiler (GCC) abstürzen:

```
linux$ gcc-2.95.2 -0 bug.c  
gcc: Internal error: program cc1 got fatal signal 11
```

*Welcher Fehler verursacht dieses Fehlschlagen?*



# Fehler

---



Was ist der Fehler in GCC?

Ein **Fehler** ist eine Abweichung vom Korrekten, Richtigen, oder Wahren.

— IEEE Standard Glossary of SE Terminology

Um zu zeigen, dass etwas ein Fehler ist, müssen wir *die Abweichung zeigen*:

- *einfach* für das konkrete Fehlschlagen
- *schwierig* für den Programmcode

Ansatz: *Deduktion* – Schließen vom abstrakten (Code) auf das konkrete (Ablaufen): Statische Analyse, Verifikation, ...

*Wo weicht GCC ab – und wovon?*



# Ursachen

---



Was ist die Ursache des GCC-Absturzes?

Eine **Ursache** eines Ereignisses („**Wirkung**“) ist ein vorangegangenes Ereignis, ohne dies das Ereignis nicht aufgetreten wäre.

— Microsoft Encarta

Um Kausalität zu zeigen, müssen wir zeigen, dass

1. **die Wirkung auftritt, wenn die Ursache auftritt**
2. **die Wirkung *ausbleibt*, wenn die Ursache *ausbleibt*.**

Allgemeiner Ansatz: *Experimentieren* – Konstruktion einer *Theorie* aus einer Folge von Experimenten (Läufen)

*Können wir Experimentieren automatisieren?*



# Isolieren von Fehlerursachen



Mit systematischem Testen können wir die fehlerverursachende Eingabe isolieren:

#	GCC-Eingabe	Test
1	<code>double bug(...) { int i, j; i = 0; for (...) { ...} ...}</code>	✗
5	<code>double bug(...) { int i, j; i = 0; for (...) { ...}...}</code>	✗
⋮		
19	<code>... z[i] = z[i] * (z[0] + 1.0); ...</code>	✗
18	<code>... z[i] = z[i] * (z[0] + 1.0); ...</code>	✓
⋮		
4	<code>double bug(...) { int i, j; i = 0; for (...) { ...} ...}</code>	✓
3	<code>double bug(...) { int i, j; i = 0; for (...) { ...} ...}</code>	✓
2	<code>double bug(...) { int i, j; i = 0; for (...) { ...} ...}</code>	✓

+ 1.0 ist Fehlerursache – nach nur 19 Tests (≈ 2 Sekunden)



# Delta Debugging

---



5/22

Delta Debugging isoliert *fehlerverursachende Unterschiede*:

**Eingaben.** Der Mozilla-Browser stürzt beim Drucken ab —  
Ursache: 1 von 900 HTML-Zeilen (<SELECT>)

A. Zeller + R. Hildebrandt, *Simplifying and Isolating Failure-Inducing Input*, IEEE TSE 28(2), 2002

**Code-Änderungen.** DDD läuft nicht mehr mit GDB 4.17 —  
Ursache: 1 von 8.721 Änderungen (infcmd.c:1239).

A. Zeller, *Yesterday, my program worked. Today, it does not. Why?*, Proc. ACM ESEC/FSE 1999

**Threads.** *Data Race* in parallelem Raytracer —

Ursache: 1 von 3,8 Mrd. Thread-Wechseln (Scene.java:91)

J.-D. Choi + A. Zeller, *Isolating Failure-Inducing Thread Schedules*, Proc. ACM ISSTA 2002

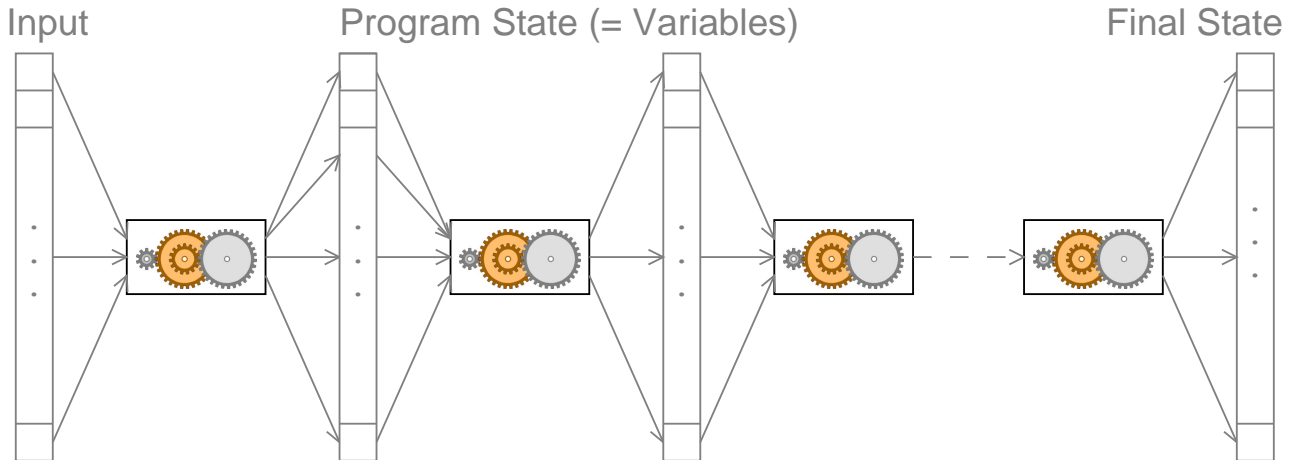
Vollautomatisches, rein testbasiertes Verfahren.



# Was passiert in GCC?



Der Unterschied + 1.0 ist erst der Anfang einer *Ursache-Wirkungs-Kette* im GCC-Lauf.

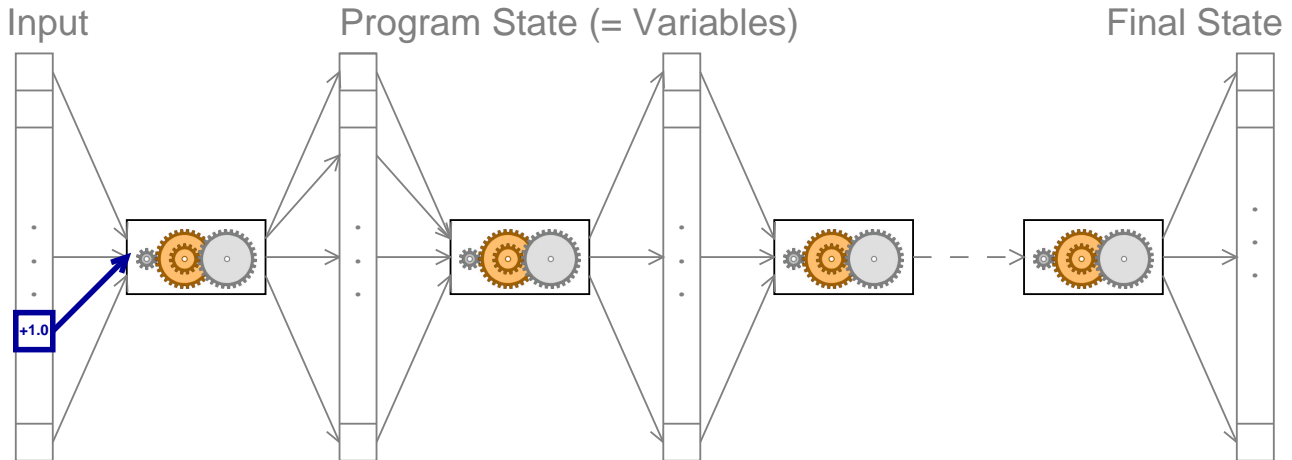


# Was passiert in GCC?



6/22

Der Unterschied  $+ 1.0$  ist erst der Anfang einer *Ursache-Wirkungs-Kette* im GCC-Lauf.

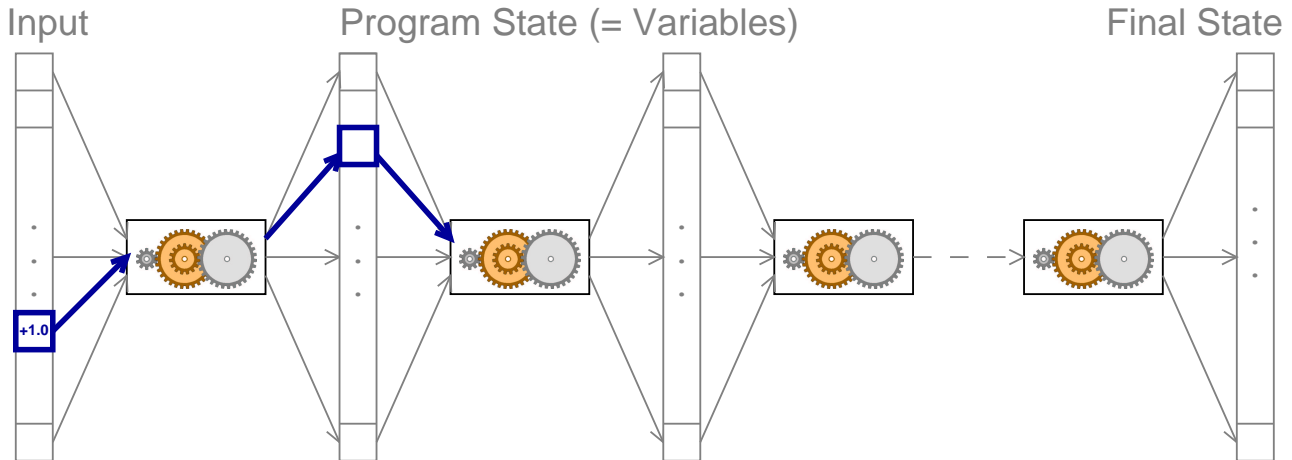




# Was passiert in GCC?



Der Unterschied `+ 1.0` ist erst der Anfang einer *Ursache-Wirkungs-Kette* im GCC-Lauf.

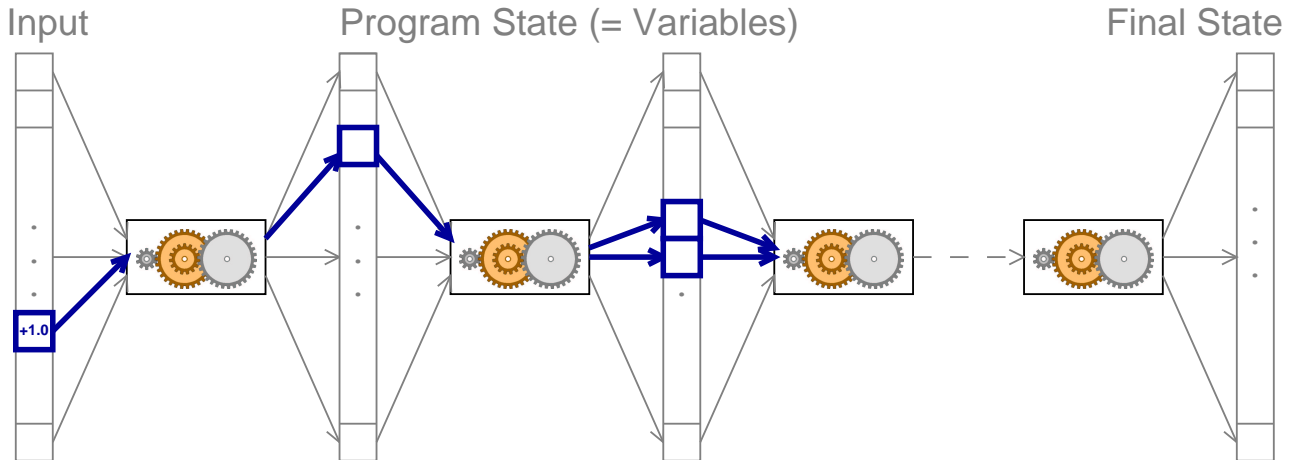


# Was passiert in GCC?



6/22

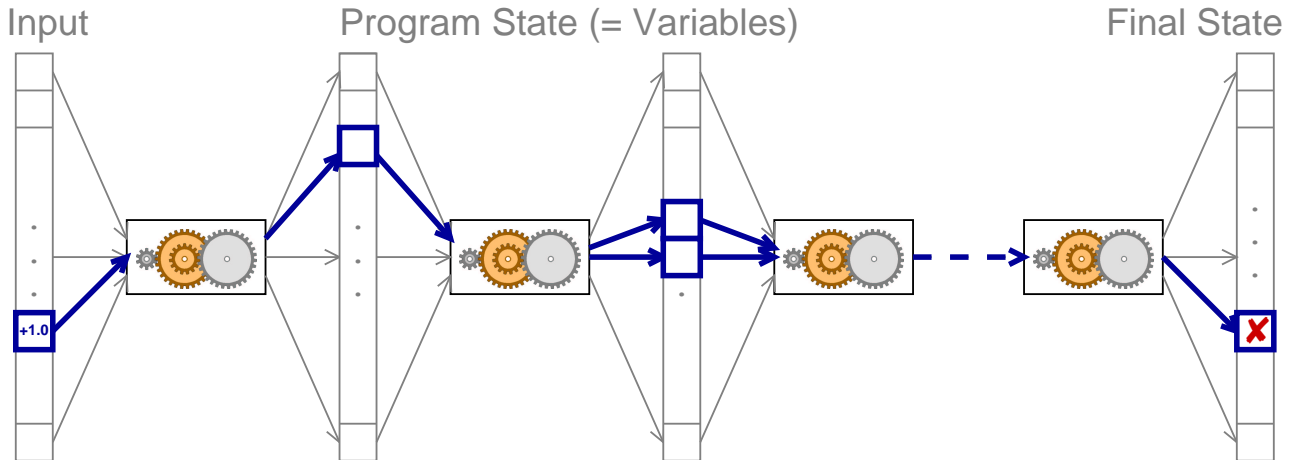
Der Unterschied  $+ 1.0$  ist erst der Anfang einer *Ursache-Wirkungs-Kette* im GCC-Lauf.



# Was passiert in GCC?



Der Unterschied  $+ 1.0$  ist erst der Anfang einer *Ursache-Wirkungs-Kette* im GCC-Lauf.



Um den Fehler zu reparieren, müssen wir diese Kette *brechen*.





# Datenströme verfolgen

---

Mit klassischer *Programmanalyse* kann man verfolgen, wie sich Daten in Programmen fortpflanzen.

Voraussetzung: Vollständiges Wissen über gesamten Code und seine Semantik ⇒ gut für kleine, abgeschlossene Programme. ■

Aber: Echte Programme sind *opak, parallel, verteilt, dynamisch, mehrsprachig* ■ – oder einfach obskur:

```
struct foo {           // Allocate string
    int tp, len;       int len = 200;
    union {           int bytes = len + 2 * sizeof(int);
        char    c[1];   foo *x = (foo *)malloc(bytes);
        int     i[1];   x->tp = STRING;
        double  d[1];   x->len = len;
    }                 strncpy(x->c, "Some string", len);
} ■
```

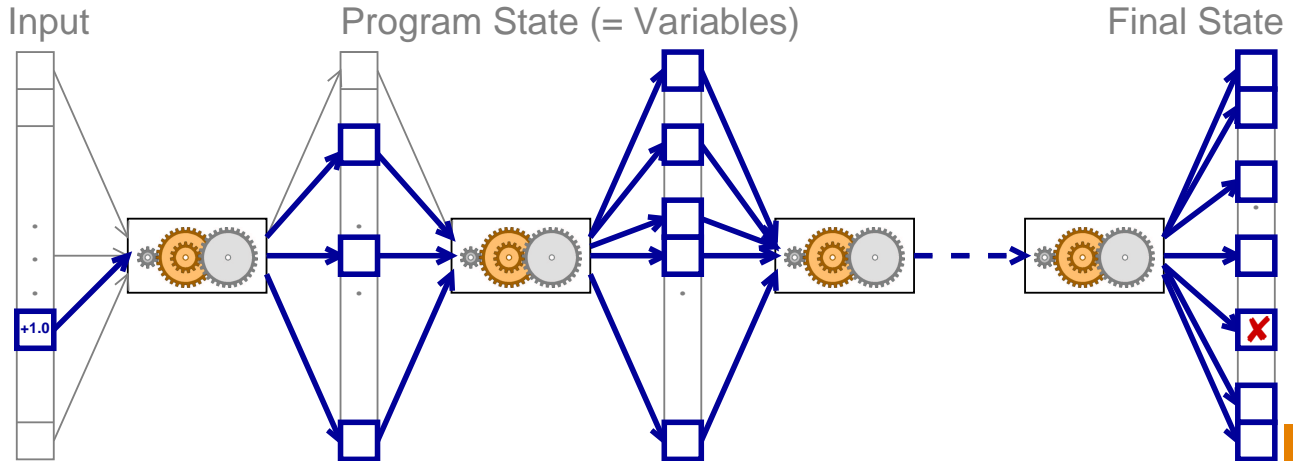


# Kleine Ursache, große Wirkung



8/22

Auswirkungen sammeln sich während der Ausführung an:



Ansatz: *relevante Zustandsunterschiede* bestimmen!





# Relevante Zustandsunterschiede

Mit einem Debugger (GDB) können wir den Zustand beobachten und verändern.

Beispiel: GCC-Zustand in der Funktion *combine\_instructions*

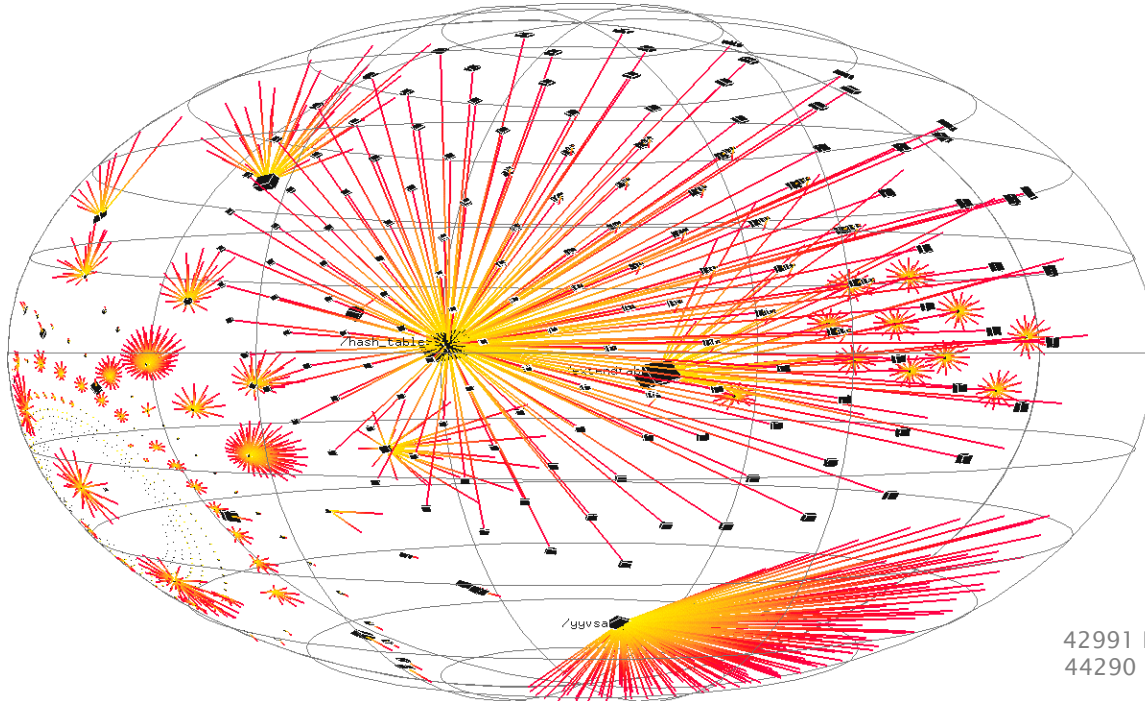
#	reg_rtx_no	cur_insn_uid	last_lineno	first_loop_store_insn	test
1	32	74	15	0x81fc4e4	✘
			?		
5	32	74	15	0x81fc4a0	✔
4	32	74	14	0x81fc4e4	?
3	32	74	14	0x81fc4a0	✔
2	31	70	14	0x81fc4a0	✔

Folge: *Strukturelle Unterschiede* bestimmen und anwenden!



# Der GCC-Speichergraph

IGOR-Prototyp betrachtet Programmzustand als *Graph*:  
Knoten sind *Variablen*, Kanten *Verweise*



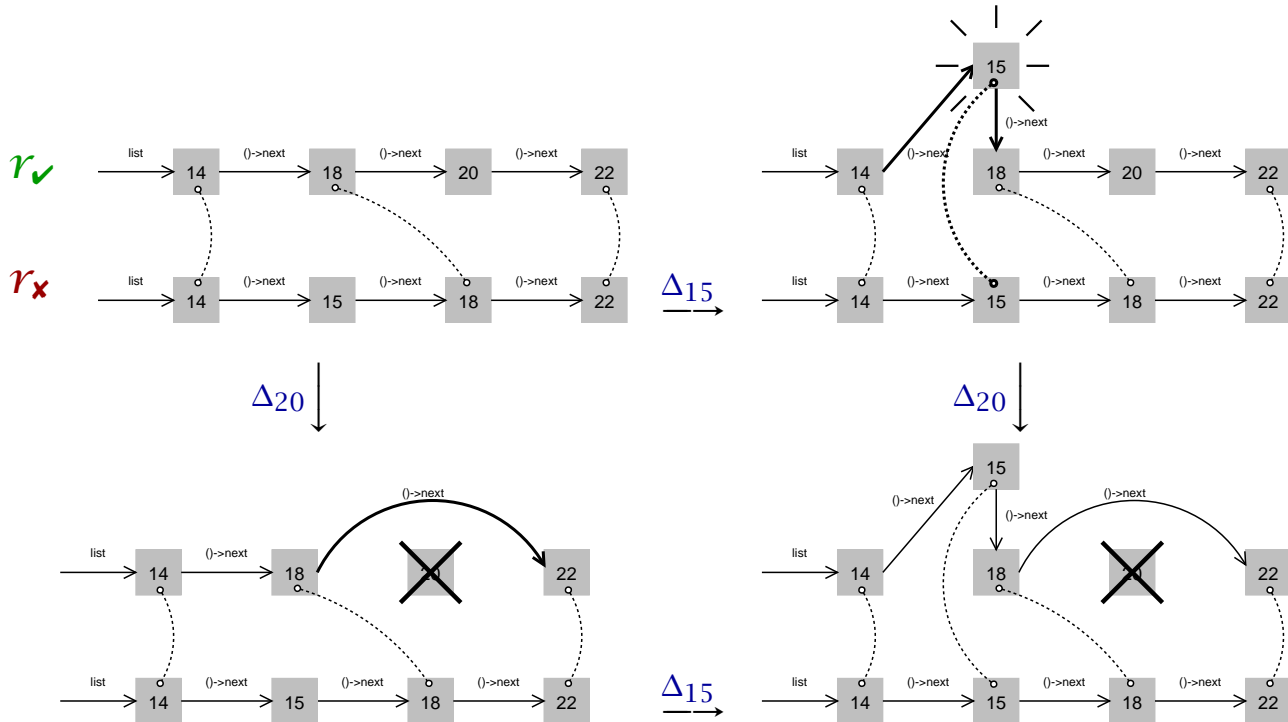
42991 Knoten  
44290 Kanten

# Strukturelle Unterschiede



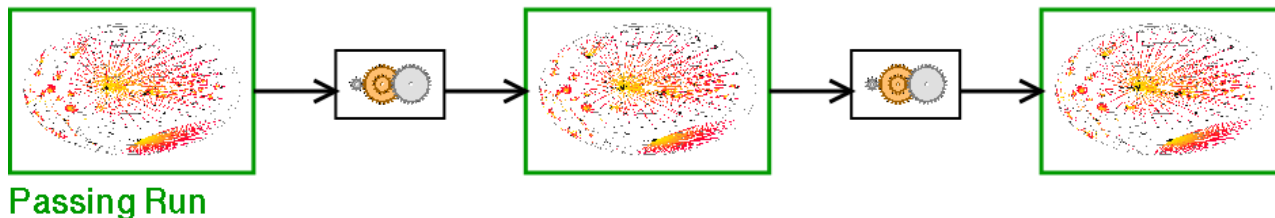
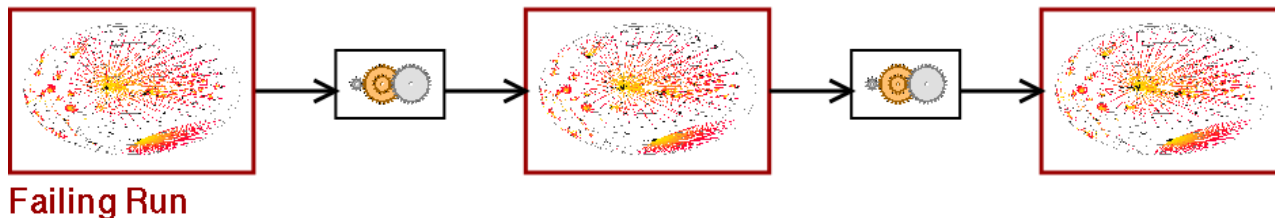
IGOR kann strukturelle Graph-Unterschiede berechnen:

$\Delta_{15}$  erzeugt eine Variable,  $\Delta_{20}$  löscht eine andere

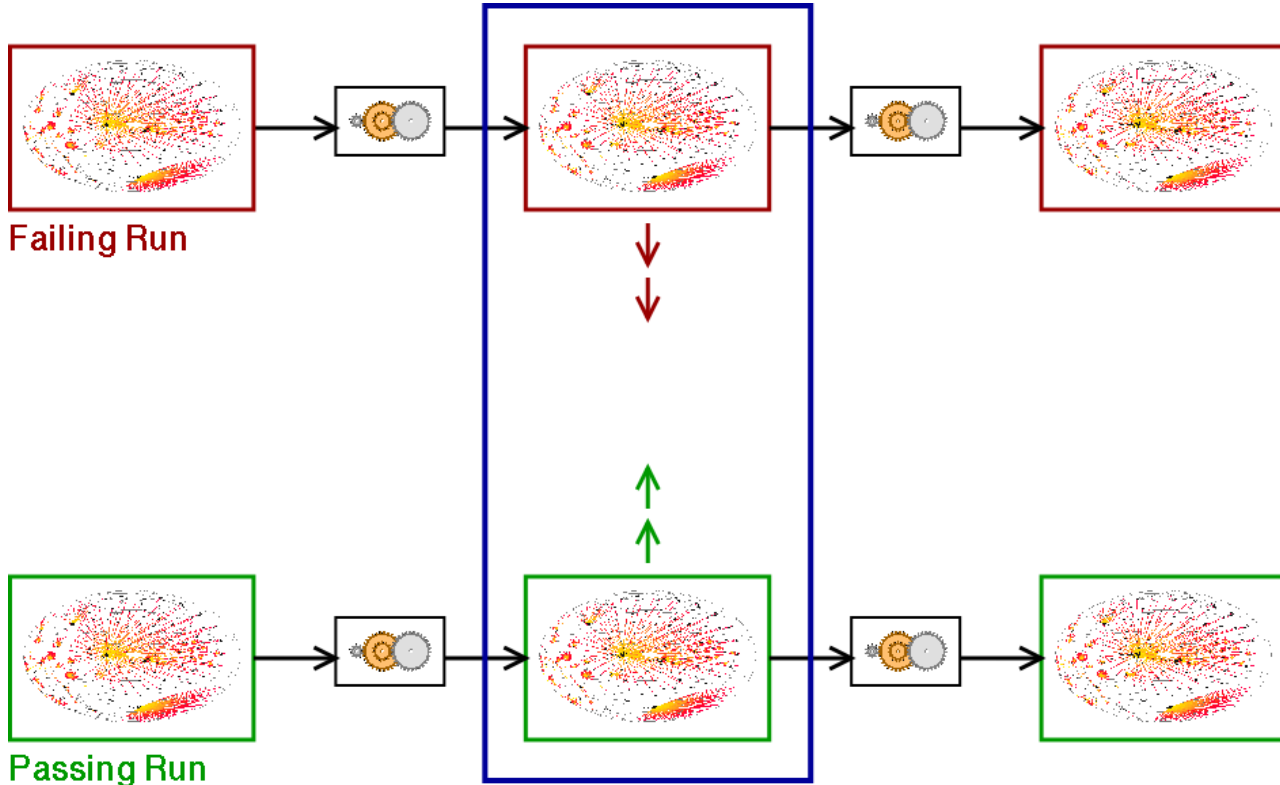




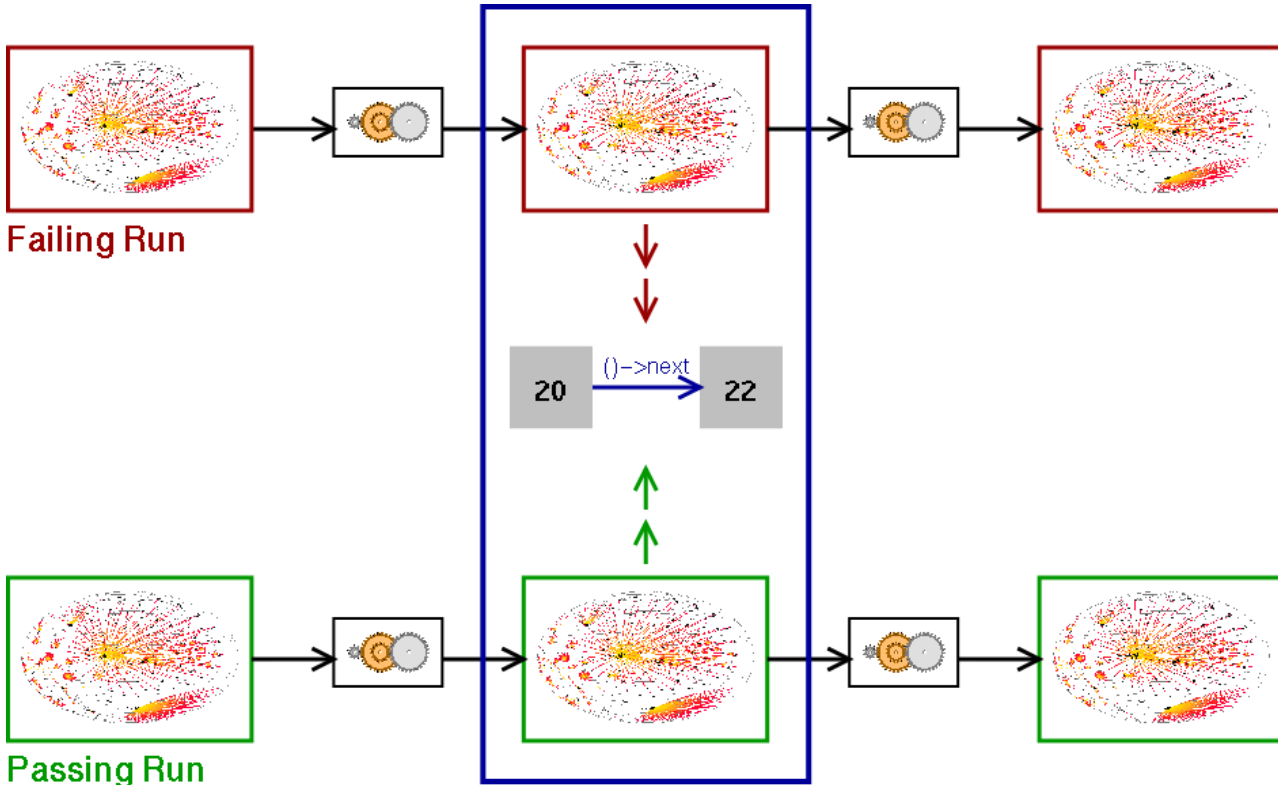
# Das Verfahren in Kürze



# Das Verfahren in Kürze



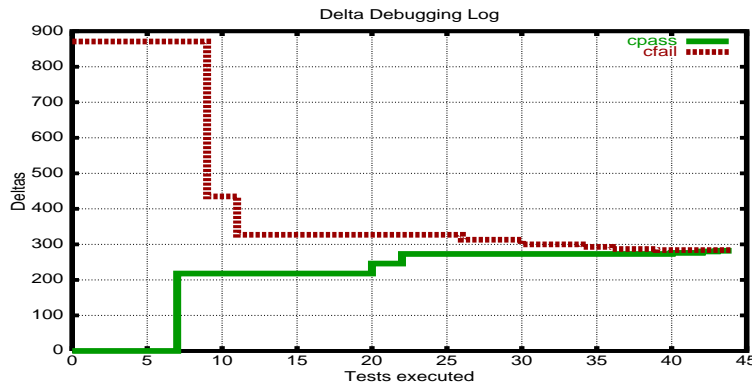
# Das Verfahren in Kürze





# Relevante Zustandsunterschiede

IGOR untersucht den Zustand von cc1 in *combine\_instructions*:  
871 Knoten (= Variablen) sind unterschiedlich:



Nur eine Variable verursacht das Fehlschlagen:

```
$m = (struct rtx_def *)malloc(12)
$m->code = PLUS
first_loop_store_insn->fld[1]...rtx = $m
```



# Die GCC Ursache-Wirkungs-Kette



Nach 59 Tests hat IGOR diese Fehlerursachen isoliert:

1. Execution reaches **main**.

Since the program was invoked as “cc1 -0 fail.i”, variable **argv[2]** is now “fail.i”.

2. Execution reaches **combine\_instructions**.

Since **argv[2]** was “fail.i”, variable **\*first\_loop\_store\_insn**→**fld[1].rtx**→**fld[1].rtx**→**fld[3].rtx**→**fld[1].rtx** is now **<new rtx\_def>**.

3. Execution reaches **if\_then\_else\_cond (95th hit)**.

Since **\*first\_loop\_store\_insn**→**fld[1].rtx**→**fld[1].rtx**→**fld[3].rtx**→**fld[1].rtx** was **<new rtx\_def>**, variable **link**→**fld[0].rtx**→**fld[0].rtx** is now **link**.

4. Execution ends.

Since variable **link**→**fld[0].rtx**→**fld[0].rtx** was **link**, the program now **terminates with a SIGSEGV signal**.  
The program fails.

Gesamt-Laufzeit: 6 Sekunden (+ 90 Minuten GDB-Overhead)





# Fehler isolieren

---

Wir können den **Fehler** eingrenzen, indem wir (von Hand) **fehlerhafte** und **nicht-fehlerhafte** Ursachen unterscheiden.



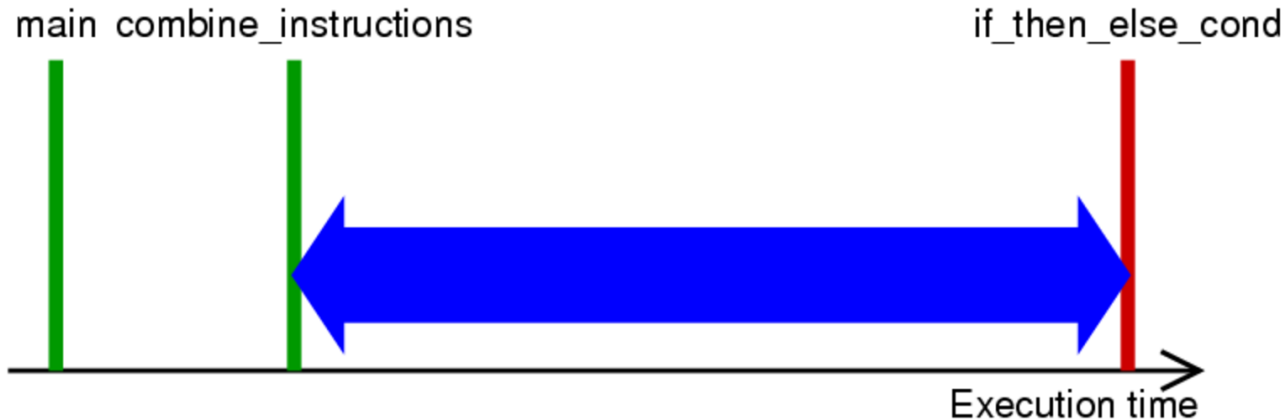
# Fehler isolieren

---



14/22

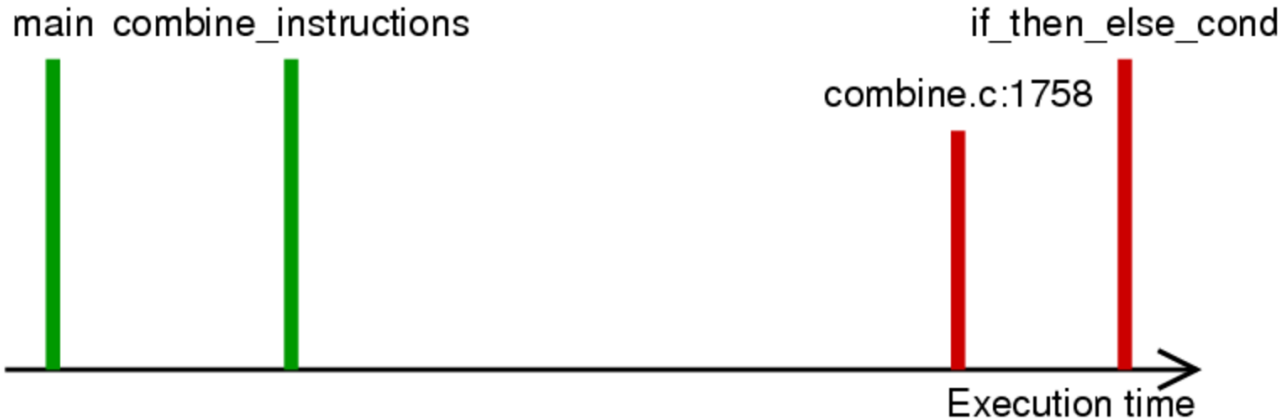
Wir können den **Fehler** eingrenzen, indem wir (von Hand) **fehlerhafte** und **nicht-fehlerhafte** Ursachen unterscheiden.



# Fehler isolieren



Wir können den **Fehler** eingrenzen, indem wir (von Hand) **fehlerhafte** und **nicht-fehlerhafte** Ursachen unterscheiden.

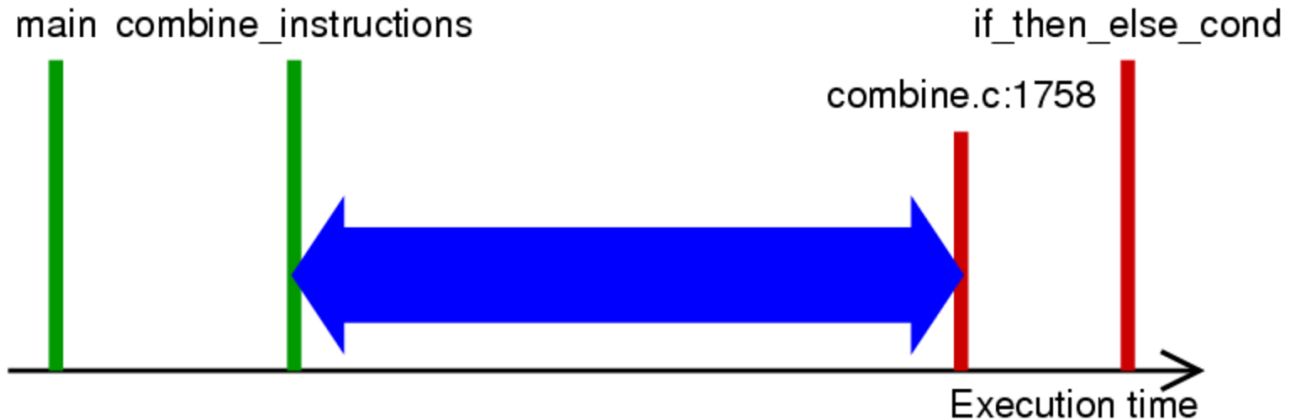




# Fehler isolieren



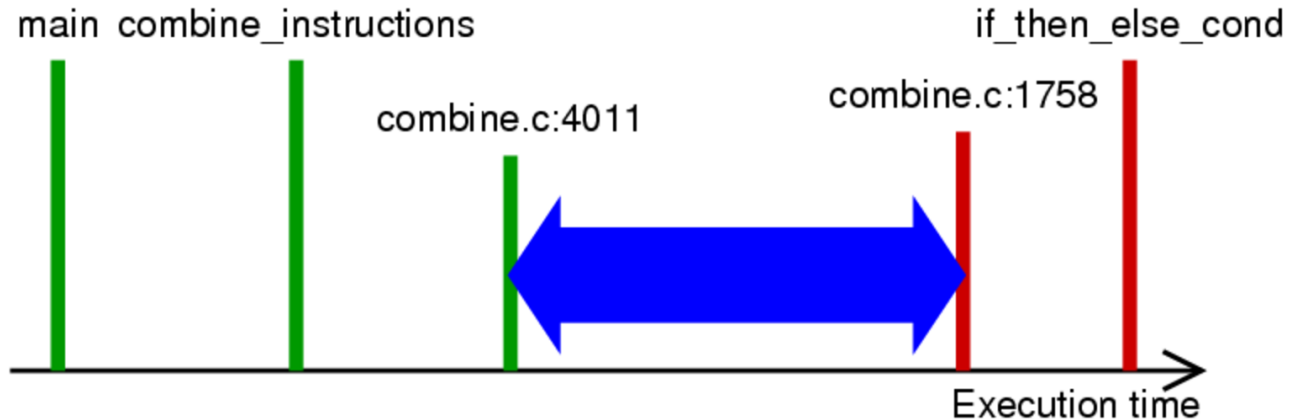
Wir können den **Fehler** eingrenzen, indem wir (von Hand) **fehlerhafte** und **nicht-fehlerhafte** Ursachen unterscheiden.





# Fehler isolieren

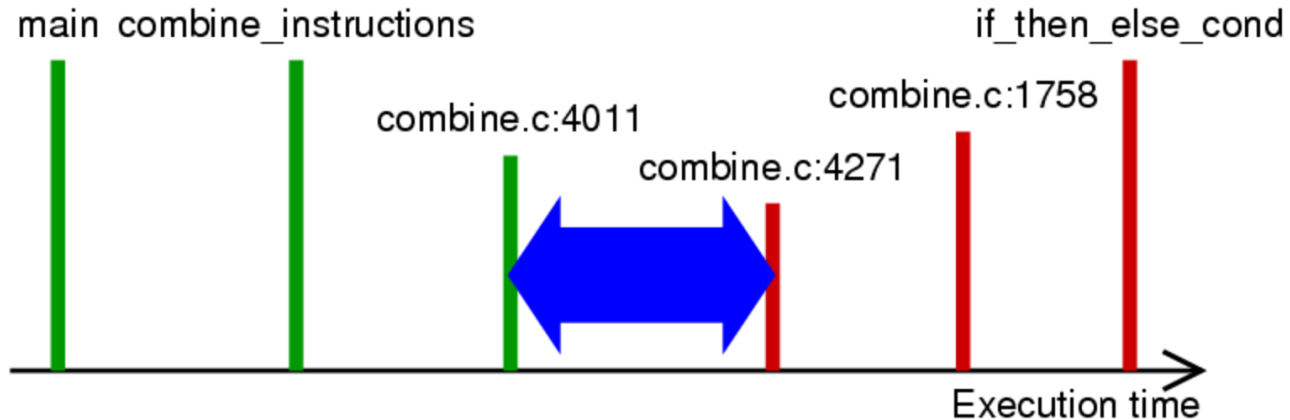
Wir können den **Fehler** eingrenzen, indem wir (von Hand) **fehlerhafte** und **nicht-fehlerhafte** Ursachen unterscheiden.





# Fehler isolieren

Wir können den **Fehler** eingrenzen, indem wir (von Hand) **fehlerhafte** und **nicht-fehlerhafte** Ursachen unterscheiden.



Falsches Alias im Distributivgesetz in Zeilen 4013–4019;  
behoben in 2.95.3

$$(+ (* a b) c) \Rightarrow (* (+ a c_1)(+ b c_2)) \text{ with } c = c_1 = c_2$$





# Herausforderungen

---

## Wie finden wir relevante Ereignisse?

*Warum z.B. bei `combine_instructions` suchen?*

Heute: Suchen im *Backtrace* des fehlschlagenden Laufes

Morgen: Fokus auf *Ursachen-Übergänge*.■

## Wie erfassen wir C-Zustände akkurat?

*Zeigt  $p$  auf etwas, und wenn ja, wieviele?*

Heute: Anfragen an das Laufzeitsystem + Heuristiken.

Morgen: Wechsel zu Java :-)■

## Woher wissen wir, dass ein Fehlschlag der Fehlschlag ist?

*Können Änderungen nicht neue Fehlschläge bewirken?*

Heute: Backtraces müssen übereinstimmen.

Morgen: Ausgabe, Zeit, Code-Abdeckung vergleichen.■

## Und schließlich: Wann funktioniert dies eigentlich?



Programm einreichen



Aufrufe angeben



Auf „Debug it“ klicken



Diagnose per e-mail

600 Einreichungen  
seit Dezember 2003

56% „pinpoints the bug“  
22% „helpful insights“

Open Source!



# Delta Debugging Plug-Ins



18/22

The screenshot shows the Eclipse IDE interface. The Package Explorer on the left shows a project named 'IPLproject' with a package 'edu.marlbora.msie.ipl' containing a file 'AClass.java'. The main editor window displays the source code for 'AClass.java'. The code includes a package declaration, a multi-line comment, and a public class 'AClass' with a 'main' method that prints 'I am a class'. A 'Delta Debugging' dialog box is overlaid on the code, with the text 'Determining failure cause, please stand by...' and a 'Cancel' button. The Outline view on the right shows the class structure with 'AClass' and its 'main' method. The Tasks view at the bottom is empty. The status bar at the bottom right shows 'Writable', 'Insert', and '14 : 4'.

```
package edu.marlbora.msie.ipl;

/**
 * @author gc-admin
 * To change this generated comment edit the template
 * Window>Preferences>Java>Templates.
 * To enable and disable the creation of type comments
 * Window>Preferences>Java>Code Generation.
 */
public class AClass {
    public static void main(String[] args) {
        System.out.println("I am a class");
    }
}
```

Delta Debugging

Determining failure cause, please stand by...

Cancel

C	!	Description	Resource	In Folder	Location

Package Explorer Hierarchy

Writable Insert 14 : 4

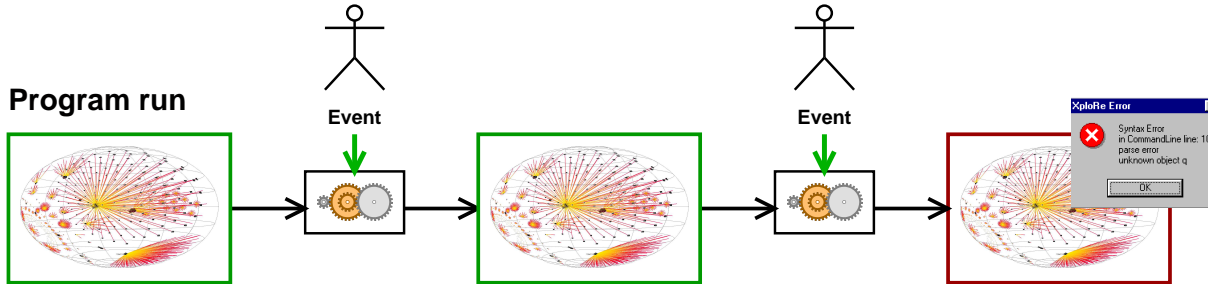


# Delta Debugging in einem Lauf



19/22

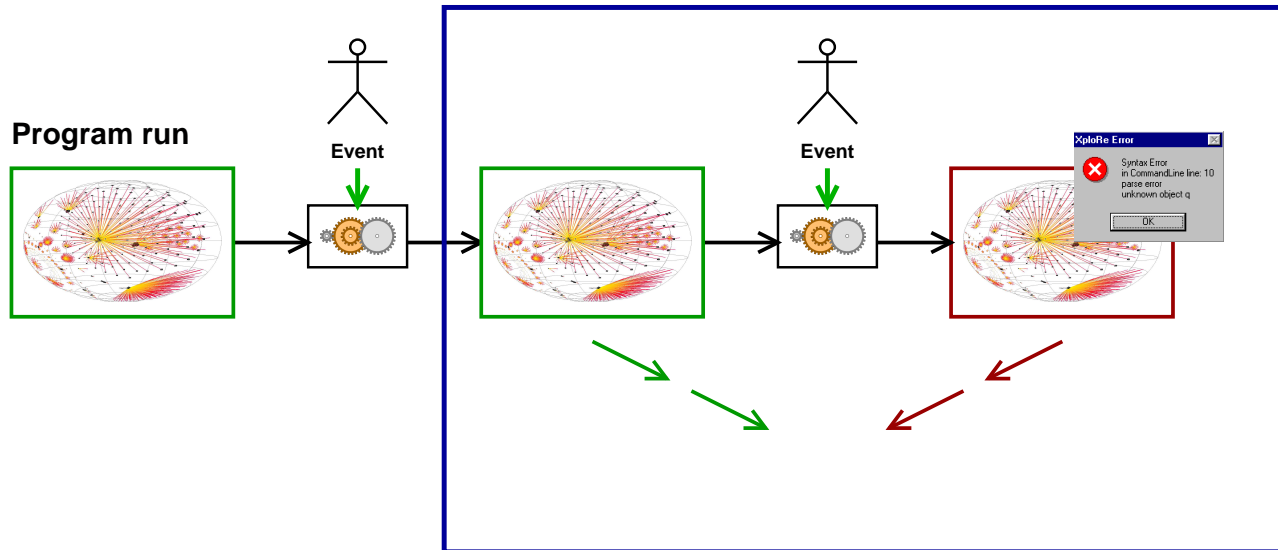
In einem reaktiven Programm kann ein Lauf genügen:



# Delta Debugging in einem Lauf



In einem reaktiven Programm kann ein Lauf genügen:



Vergleich der Zustände zu *verschiedenen Zeitpunkten* zeigt Unterschiede...

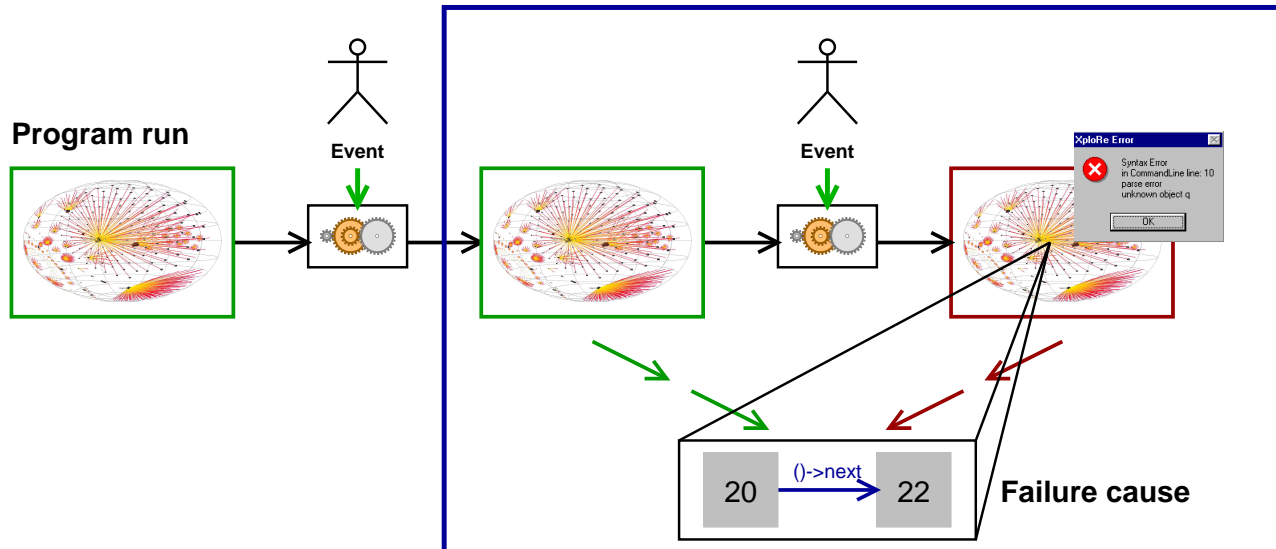




# Delta Debugging in einem Lauf



In einem reaktiven Programm kann ein Lauf genügen:



Vergleich der Zustände zu *verschiedenen Zeitpunkten* zeigt Unterschiede, die zu Ursachen eingengt werden können.

Anwendungen: interaktive Programme, Server, Treiber...



# Selbstheilende Programme



The image shows three overlapping Windows-style dialog boxes. The top box, titled "Oops", features a red "X" icon and the text: "A fatal error has occurred. Auto-repair in progress; please stand by...". The middle box, titled "Auto-repair successful", features a yellow warning triangle with a black exclamation mark and the text: "Auto-repair successful. Some services have been turned off. (Details)". The bottom box, titled "Auto-repair details", features a blue information icon and the text: "Sub-Pixel anti-aliasing has been turned off, as it caused a fatal error. (Reactivate)". An "OK" button is visible at the bottom of the "Auto-repair details" box.



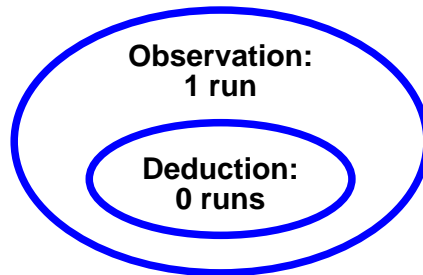
# Rückblick und Ausblick

---



21/22

Letzte 20 Jahre: *Deduktion* und *Observations*-Techniken

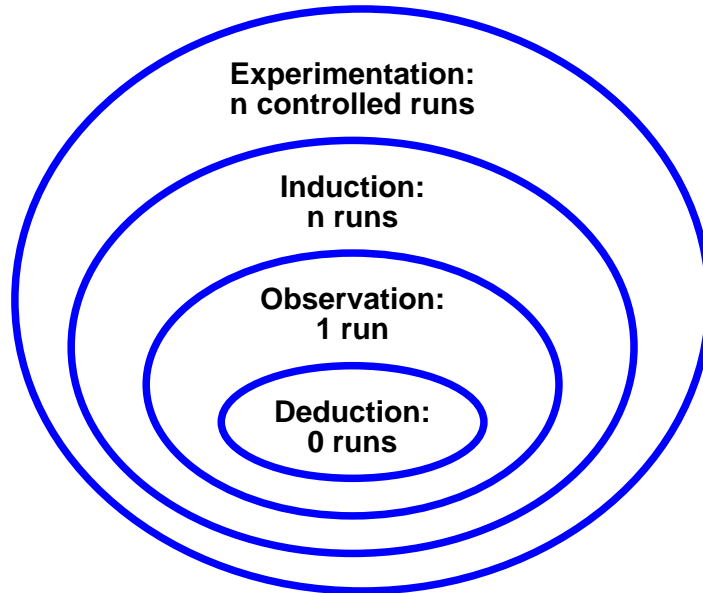


# Rückblick und Ausblick

---



Letzte 20 Jahre: *Deduktion* und *Observations*-Techniken



Nächste 20 Jahre: *Induktion* und *Experimentieren*?





# Fazit

---

- ★ Wir können die Abwesenheit von Fehlern beweisen – doch nie die *Abwesenheit von Überraschungen*.
- ★ Fehlerursachen können *automatisch isoliert werden...*
  - mit Hilfe eines automatischen Tests
  - in dem wenigstens ein Testfall erfolgreich ist
- ★ Systematisches *Experimentieren* kann „klassische“ Programm-Analyse erheblich verbessern.
- ★ Durch Automatisierung wird Fehlersuche zu einer *verstandenen und systematischen Disziplin*.
- ★ Buch „Why does my program fail?“ (MK) im Mai 2005

<http://www.askigor.org/>



# Zum Weiterlesen

---

**Why does my Program Fail? A Guide to Automated Debugging.** Morgan Kaufmann Publishers, Spring 2005.

**Isolating Cause-Effect Chains from Computer Programs.** Proc. ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2002), Charleston, Nov. 2002.

**Isolating Failure-Inducing Thread Schedules.** (w/ J.-D. Choi) Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002), Rom, July 2002.

**Simplifying and Isolating Failure-Inducing Input.** (w/ R. Hildebrandt) IEEE Transactions on Software Engineering 28(2), February 2002, pp. 183–200.

**Automated Debugging: Are We Close?** IEEE Computer, Nov. 2001, pp. 26–31.

**Visualizing Memory Graphs.** (w/ T. Zimmermann) Proc. of the Dagstuhl Seminar 01211 „Software Visualization“, May 2001. LNCS 2269, pp. 191–204.

**Yesterday, my program worked. Today, it does not. Why?** Proc. ACM SIGSOFT Conference (ESEC/FSE 1999), Toulouse, Sep. 1999, LNCS 1687, pp. 253–267.

<http://www.askigor.org/>





# About this Presentation

---

This presentation was created by Andreas Zeller, Professor of Computer Science at Saarland University, Saarbrücken, Germany. Contact him at

<http://www.st.cs.uni-sb.de/~zeller/>

This presentation, its source code, and additional material can be downloaded at

<http://www.st.cs.uni-sb.de/papers/fse2002/>

This presentation is licensed under the Creative Commons Attribution License. To view a copy of this license, visit

<http://creativecommons.org/licenses/by/1.0/>

or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

