Karlsruhe Institute of Technology

Localizing Defects in Multithreaded Programs by Mining Dynamic Call Graphs

Frank Eichinger, Victor Pankratius , Philipp Große, Klemens Böhm

Institute for Program Structures and Data Organization (IPD)

Foto: Stadt Karlsruhe

Defects in Software

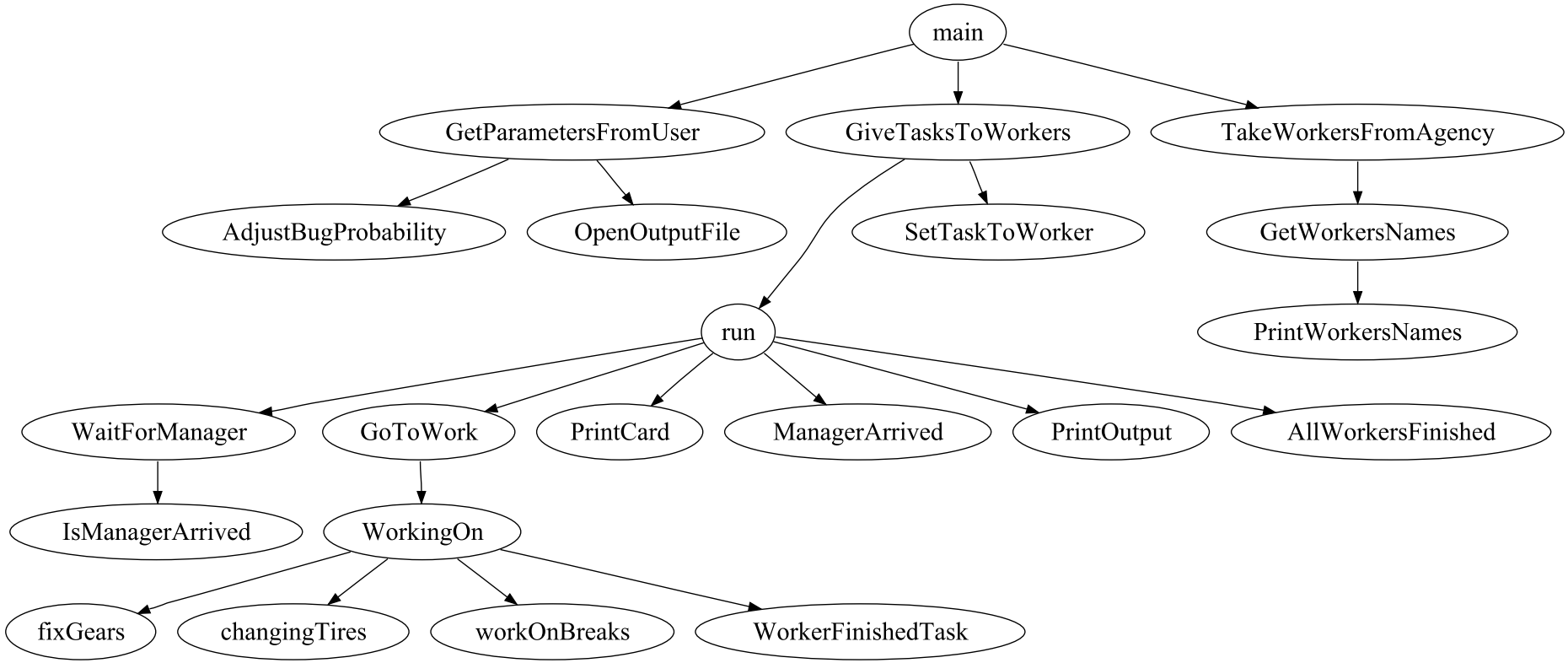
- Software is rarely free from defects.
- In the software-development process,
  - at least 35% of the time is spent for debugging activities,
    and
  - defect localisation is the most difficult task.

⇒ Defects lead to huge costs!

- Automated means for **defect-localization** are needed**.**
- Various (data-mining-based) approaches are available.
- We focus on **Call-Graph based detection.**
  - In particular: occasional bugs

Example

- **GarageManager**: „blocking critical section“



Example – The Defect

```
void GoToWork(){
…
      switch (taskNumber % 8) {
          case 0: WorkingOn(„fix gear“, 2000);                    break;
          case 1: WorkingOn(„change tires“, 1400);                break;
          // simular for case 2 to 6 …
           case 7: WorkingOn(„work on breaks“, 2200);             break;    }
…
}
```

Example – The Failure

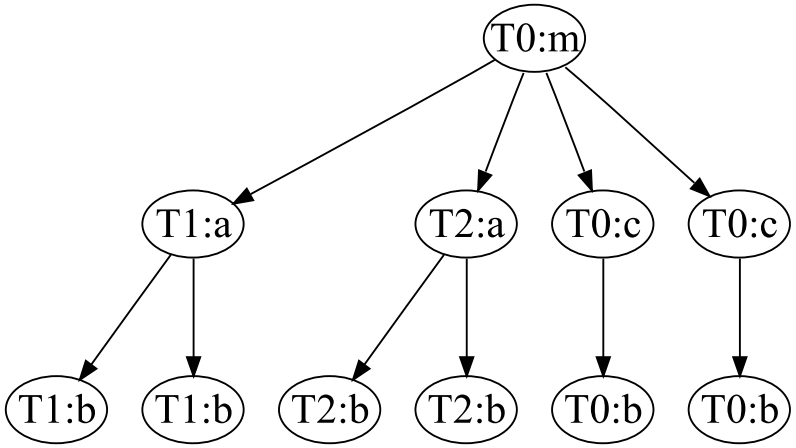
```
run(){
…
status.ManagerArrived();
boolean tasksNotFinshed = true, printedOutput = false;
while (tasksNotFinished){
    printedOutput = PrintedOutput(printOutput);
            synchronized(status){
            if (status.AllWorkersFinished())
                        tasksNotFinished = false;
            else
                        yield();}
…
}
```

Agenda

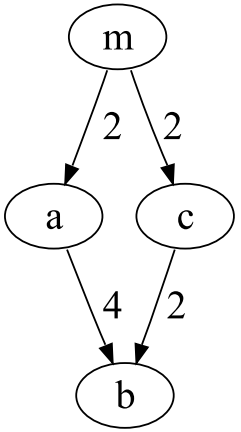
- Call-Graph Representation for Multithreading Programs
- Localizing Defects based on Call-Graphs
- Defectiveness-Likilihood Calculation
- Benchmark Results

Call-Graph Representations

unreduced

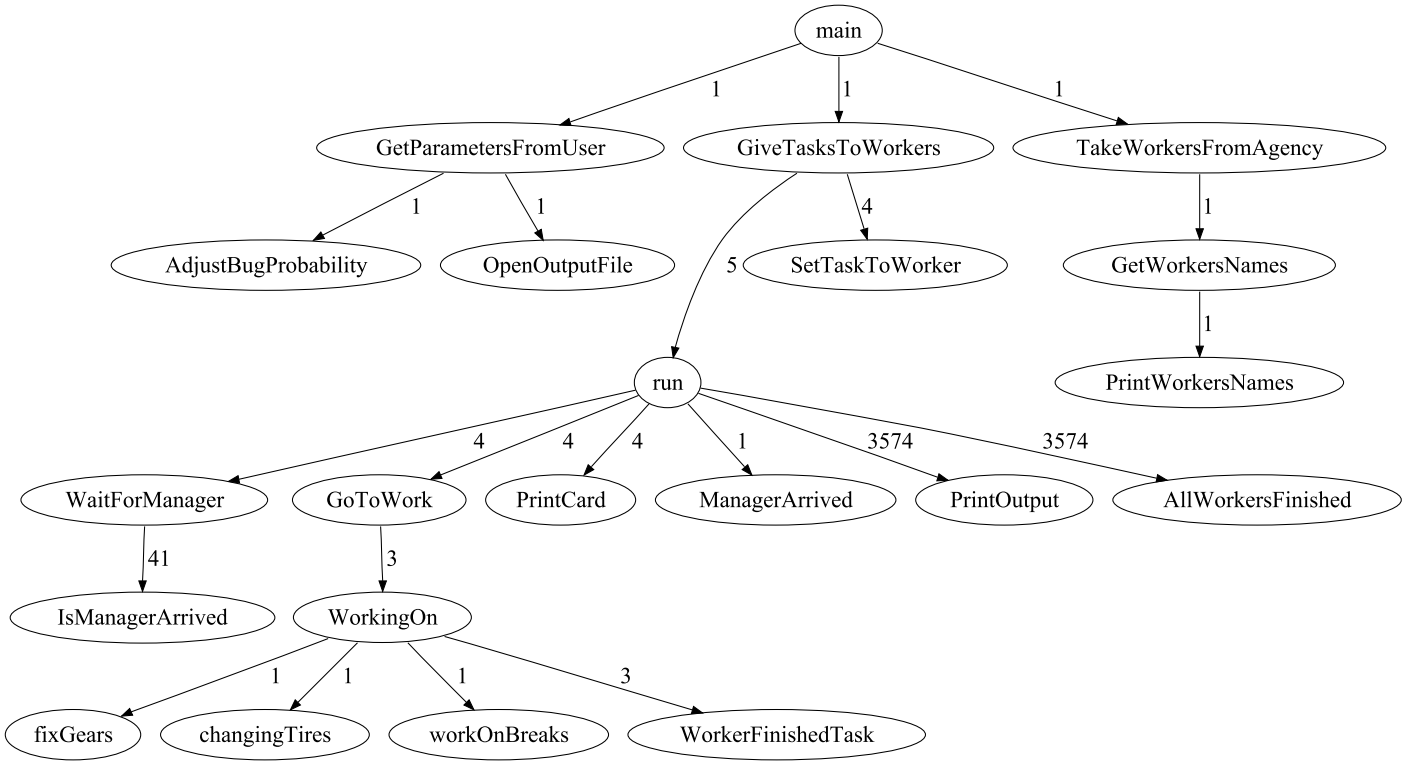


reduced



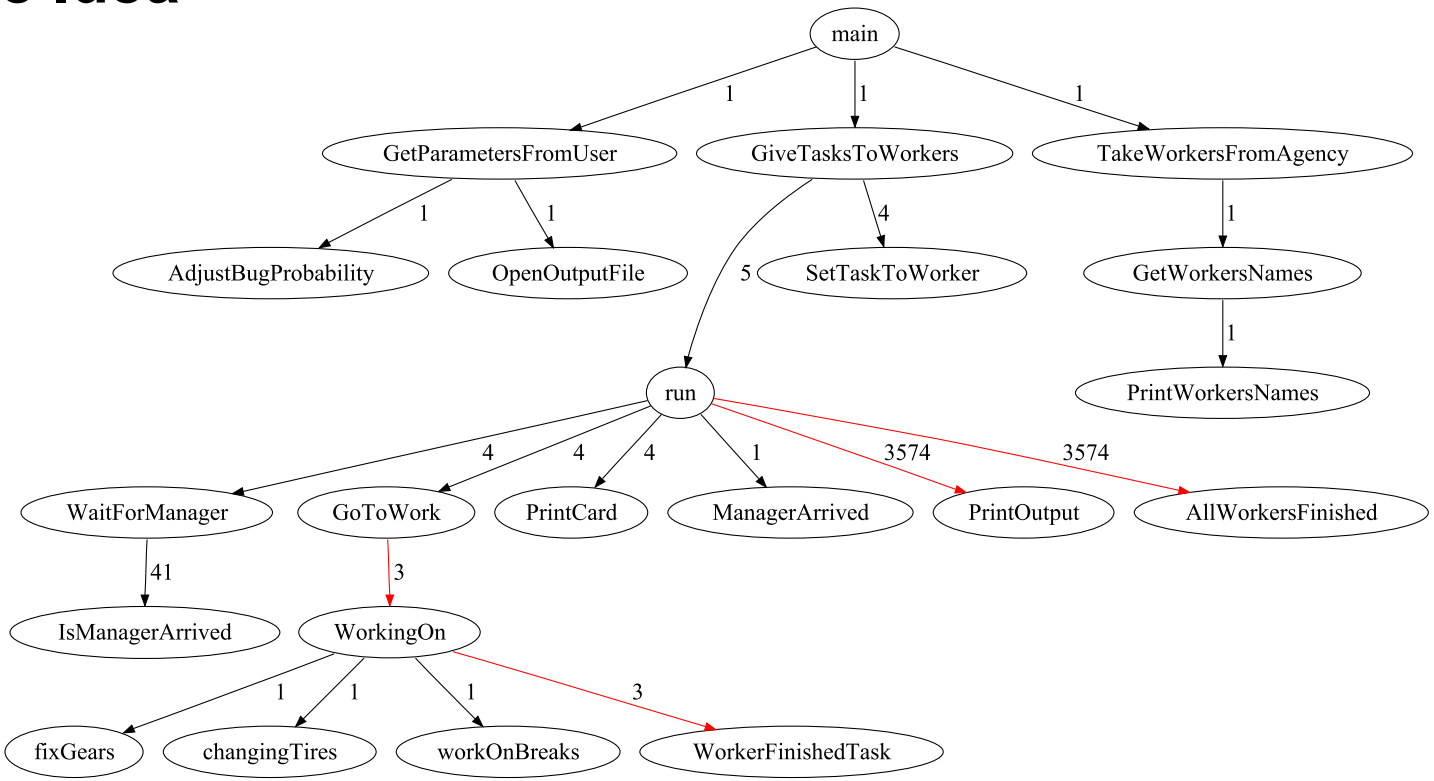
generate call-graphs using AspectJ

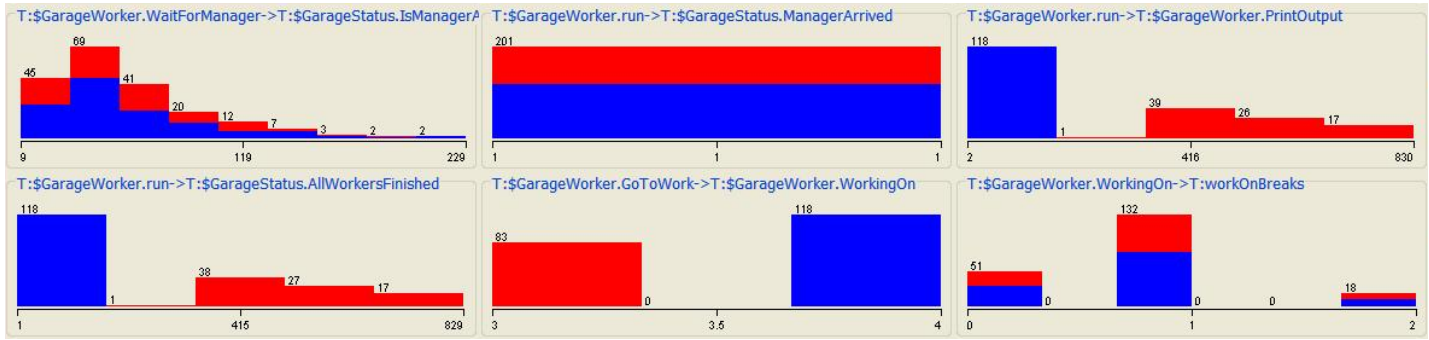
Example – cont.



	a -> b	b -> c	a -> d	…	Class
g1	445	445	7	…	failed
g2	128	256	0	…	correct
…	…	…	…	…	…

Basic Idea


main
1
1
1
GetParametersFromUser
GiveTasksToWorkers
TakeWorkersFromAgency
1
1
4
1
AdjustBugProbability
OpenOutputFile
5
SetTaskToWorker
GetWorkersNames
1
PrintWorkersNames
run
4
4
4
1
3574
3574
WaitForManager
GoToWork
PrintCard
ManagerArrived
PrintOutput
AllWorkersFinished
41
3
IsManagerArrived
WorkingOn
1
1
1
3
fixGears
changingTires
workOnBreaks
WorkerFinishedTask


T:$GarageWorker.WaitForManager->T:$GarageStatus.IsManagerA
T:$GarageWorker.run->T:$GarageStatus.ManagerArrived
T:$GarageWorker.run->T:$GarageWorker.PrintOutput
T:$GarageWorker.run->T:$GarageStatus.AllWorkersFinished
T:$GarageWorker.GoToWork->T:$GarageWorker.WorkingOn
T:$GarageWorker.WorkingOn->T:workOnBreaks

Defectiveness-Likihood Calculation

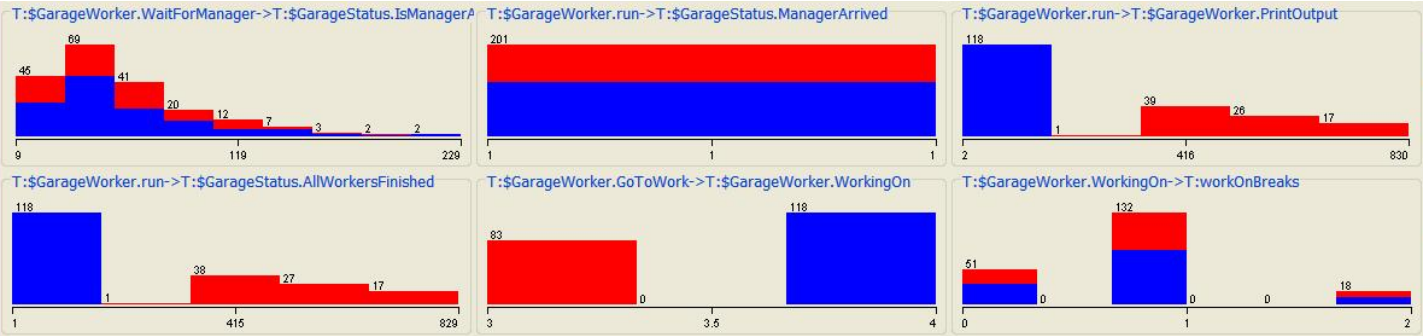
Entropy-based measure:

$$Info(D) := -\sum_{i=1}^{|\mathbb{D}_C|} \frac{|D_i|}{|D|} \cdot \log_2(\frac{|D_i|}{|D|})$$

$$InfoGain(A, D) := Info(D) - \sum_{j=1}^{|\mathbb{D}_A|} \frac{|D_j|}{|D|} \cdot Info(D_j)$$

$$SplitInfo(A, D) := -\sum_{j=1}^{|\mathbb{D}_A|} \frac{|D_j|}{|D|} \cdot \log_2(\frac{|D_j|}{|D|})$$

$$GainRatio(A, D) := \frac{InfoGain(A, D)}{SplitInfo(A, D)}$$



Ranking

- Example output:

Rank	Name	SCORE
1	a()	0.9833
2	c ()	0.9204
3	d()	0.4876
3	e()	0.4876
4	b()	0.2428

Benchmark

Program	#M	LOC	#T	Description	Defect
AllocationVector	6	133	2	Allocation of memory	two-stage access
GarageManager	30	475	4	Simulation of a garage	blocking critical section
Liveness	8	120	100	Client-server simulation	orphaned thread
MergeSort	11	201	4	Recursive Sorting	two-stage access
ThreadTest	12	101	50	CPU benchmark	blocking critical section
Tornado	122	632	100	HTTP Server	no lock
Weblech	88	802	10	Download/mirror tool	no lock / race condition

Benchmark from:
Y. Eytani and S. Ur. Compiling a Benchmark of Documented Multi-Threaded Bugs. In *Proc. Int. Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

Benchmark Result

Program	Ranking Position	%LOC to Review
AllocationVector	1	17.3%
GarageManager	1	14.2%
Liveness	1	44.2%
MergeSort	1	25.9%
ThreadTest	1	18.8%
Tornado	14	23.3%
Weblech.orig	2	23.3%
Weblech.inj	5	21.8%

only 7.1% of all methods to review
23.6% of normalized source code

Conclusion

- Summary of contributions
  - Call-Graph technique transferable for multithreading programs
  - Knowledge discovery in weighted classified graphs
  - According to benchmark only 23.6% of normalized source code has to be reviewed
- Future Work
  - Thread-specific Call-Graphs
  - Include dataflow information into graphs
  - Investigate call graphs at different levels of granularity
  - No defect-localisation approach is perfect – integration with other approaches

Questions?

Karlsruhe Institute of Technology

Thank you for your attention!

BACKUP Ranking

Rank	Name	LOC	SCORE
1	GoToWork()	40	1.0
2	WorkingOn()	25	1.0
3	run()	23	1.0
-	main()	4	0.0
-	…	…	0.0

BACKUP: Fehlerlokalisierung – allgemeine Ansätze

- Statische Ansätze (z. B. ESC/Java, RacerX)
  - Aufwändige Annotationen notwendig
  - Viele Warnungen, nur geringe Fehlereingrenzung
- Dynamischer Ansatz: happens-before-Relationen
  - Idee: Überprüfung, ob happens-before-Relation erfüllt ist
  - Dazu: Aktualisierung von logischen Vektor-Uhren
    bei Zugriff auf gemeinsame Ressource (relativ aufwändig, eine Uhr pro Faden)
  - Nur solche Fehler werden erkannt, die in einem konkreten Lauf auftreten.
- Dynamischer Ansatz: Locksets (z. B. Eraser)
  - Idee: Überwachung der Menge der Sperren für jeden Faden.
  - Bei Zugriff auf gemeinsame Ressourcen Überprüfung,
    ob eine Schnittmenge mit Sperrmenge eines anderen Fadens existiert.
  - Viele Warnungen (false positives) möglich, außerdem sehr aufwändig.
- Und andere: Model Checking, Capture-Replay, …

BACKUP: Fehlerlokalisierung – allgemeine Ansätze

- Statische Herangehensweise (Analyse des Programmcodes)
  - Heuristischer Ansatz, Identifizierung von schlechtem Programmierstil
    (viele Warnungen, oft nur geringe Eingrenzung)
  - Korrelation von Programmeigenschaften (Metriken) mit Fehlern
    (eher generelle Erkenntnisse, schwierig konkrete Fehler zu finden)

- Dynamische Herangehensweise (Analyse von Programmläufen)
  - (Teil-)Automatische Debugging-Techniken
    (Delta Debugging z.B. vereinfacht Testfälle, kann aber nicht alle Fehler lokalisieren.)
  - Statistische Auswertung von instrumentierten Variablen oder Verzweigungen
    (Welche von potentiell sehr vielen Daten sollen beobachtet werden?)
  - Betrachtung von Programm-Traces (Zeilenausführungen, Sequenzen, Graphen)
    (Betrachtung von Ausführungshäufigkeiten ist oft hilfreich, vgl. Tarantula;
    Sequenzen und Graphen sind ggf. präziser, vgl. Arbeiten am IPD Böhm)

⇒ **Es existieren viele Techniken, die sich gegenseitig ergänzen.**
Jede kann bestimmte Fehler besser oder schlechter lokalisieren.