COM SYS Communication and Distributed Systems

IVU Traffic Technologies AG

Dr. Carsten Weise, IVU Traffic Technologies AG

Testen mobiler, verteilter Netzwerke mittels symbolischer Ausführung

34. TAV, FH Aachen, 2013-02-14

About the author

- Diploma + PhD in Computer Science. Aachen University of Technology
- Nearly 40 years programming experience
- Specialized in Formal Methods for Embedded Systems
- 2 years Postdoc at Aalborg University in Kim Larsen‘s group
- 8 years integration test at Ericsson
- 4 years excellence cluster on mobile software
- 4 years consultancy of small embedded companies
- From May 2012 at IVU Traffic Technologies AG
  - One of the two major players in soft+ and hardware for public transport


- PC ICTSS 2013, A-MOST 2013, TAICPART 2013

Inhalt

- Symbolische Ausführung
- Concolic Execution
- KLEE
- KLEENET
- Weiteres zum Testen

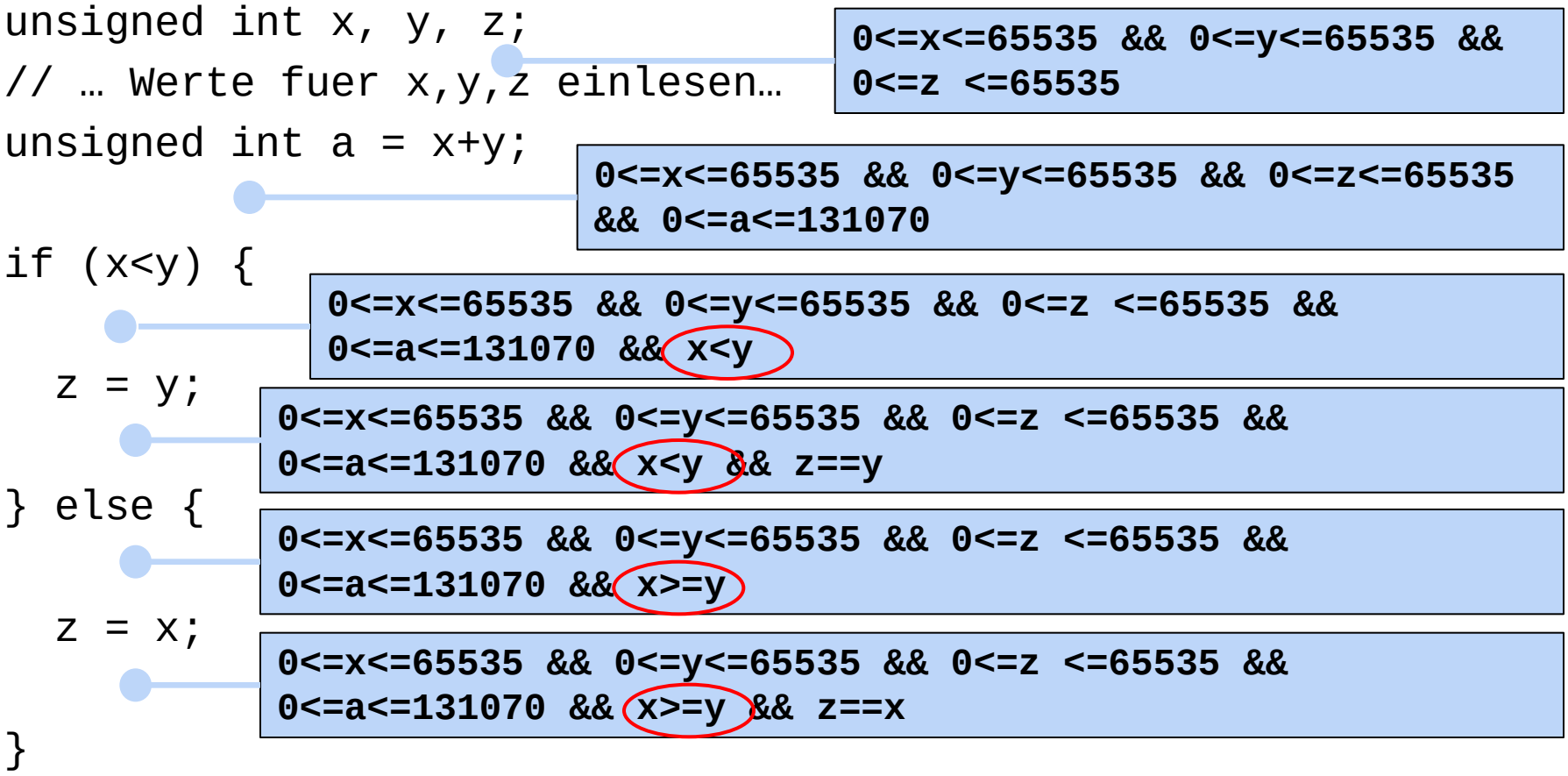
SYMBOLIC EXECUTION

Was ist Symbolische Ausführung?

- Symbolische Ausführung ist die Analyse eines Programms durch die (Simulation der) Ausführung mit symbolischen statt konkreten Werten.
- Symbolische Ausführung ist eine abstrakte Interpretation des Programmes.
- Symbolische Ausführung steht zwischen statischer Analyse und Modelchecking.

- *Symbolic Execution and Program Testing*. James C. King. IBM Thomas J. Watson Research Center, CACM 1976
- Clarke, L.A, Univ of Mass. *A System to Generate Test Data and Symbolically Execute Programs*. IEEE TSE 1976

Beispiel symbolische Ausführung



Path Constraint

Pfadabdeckung bei Symbolischer Ausführung

- Bei der symbolischen Ausführung werden alle Pfade des Programms durchlaufen
- Die Verzweigungen entlang eines Pfades führen zu *Path Constraints*
- *Path Constraints* entlang eines Pfades sind *Path Conditions*

Validierung durch Symbolische Ausführung

- Während der Symbolischen Ausführung können typische Fehler geprüft werden
  - Bereichsüberschreitungen
  - Buffer Overflow
  - Nullpointer Dereferencing
  - Zeigerfehler (je nach Detailgrad des Memory Models)
  - assertions
  - benutzerdefiniert

Testen mittels Symbolischer Ausführung

- Durch den Constraint Solver können zu einer Path Condition Variablenwerte ermittelt, die eine Testfall für den entsprechenden Pfad darstellen
- Dadurch automatische Erzeugung von Test-Suites möglich
- Coverage-Kriterien hängen vom Traversierungs-Algorithmus der Symbolischen Ausführung ab

Grenzen Symbolischer Ausführung

- Eingabewerte
- Path Constraints die der Solver nur schwer/gar nicht lösen kann (zB `y == hash(x)`)
- externe Libraries/third-party code
- Program Environment (zB Filesystem)
- exponentielles Wachstum der Pfade

CONCOLIC EXECUTION

Concolic Execution

- **concolic** = **conc**rete + symb**olic**
- symbolische Ausführung, wenn möglich
- konkrete Werte wenn nötig
- oder noch einfacher: den Anwender entscheiden lassen

```
unsigned x,y;
make_symbolic(&x, sizeof(x));
make_symbolic(&y, sizeof(y));
```

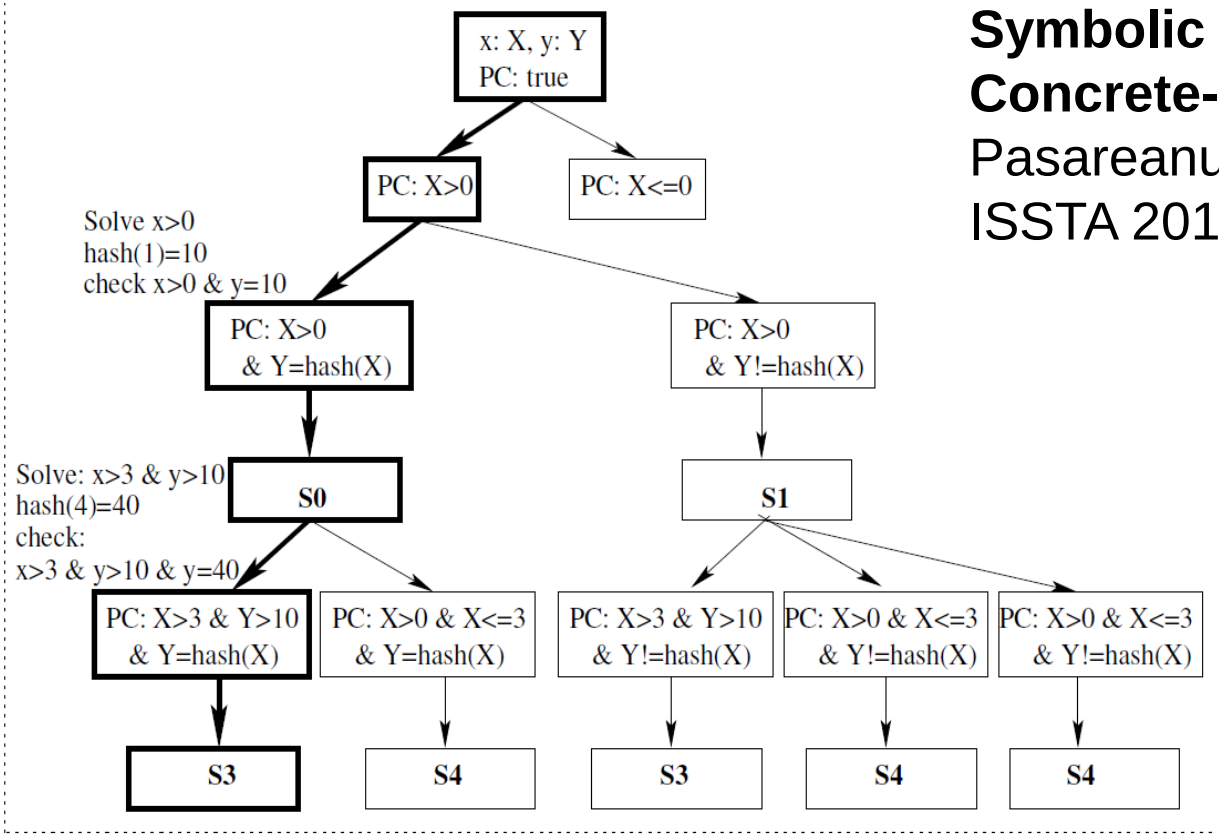
Beispiel Concolic Execution (Mixed Concrete-Symbolic Solving)

```
   @Concrete("true")
   @Partition({"x>3.0","x<=3.0"})
   double hash(double x) {...}
   void test(int x, int y) {
1:   if (x > 0) {
2:     if (y == hash(x))
3:       S0;
4:     else
5:       S1;
6:     if (x > 3 && y > 10)
7:     //if (y > 10)
8:       S3;
9:     else
10:      S4;
     }
   }
```

see:
Symbolic Execution with Mixed Concrete-Symbolic Solving
Pasareanu, Rungta, Visser.
ISSTA 2011

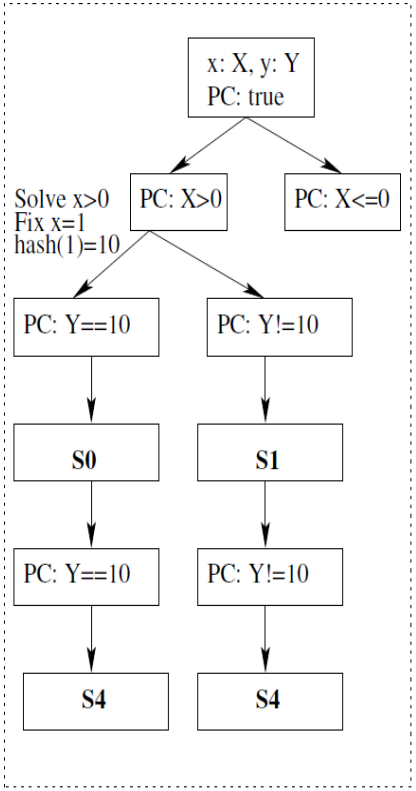
(see also `http://www.youtube.com/watch?v=azTVEwxN8zM`)

Java Pathfinder



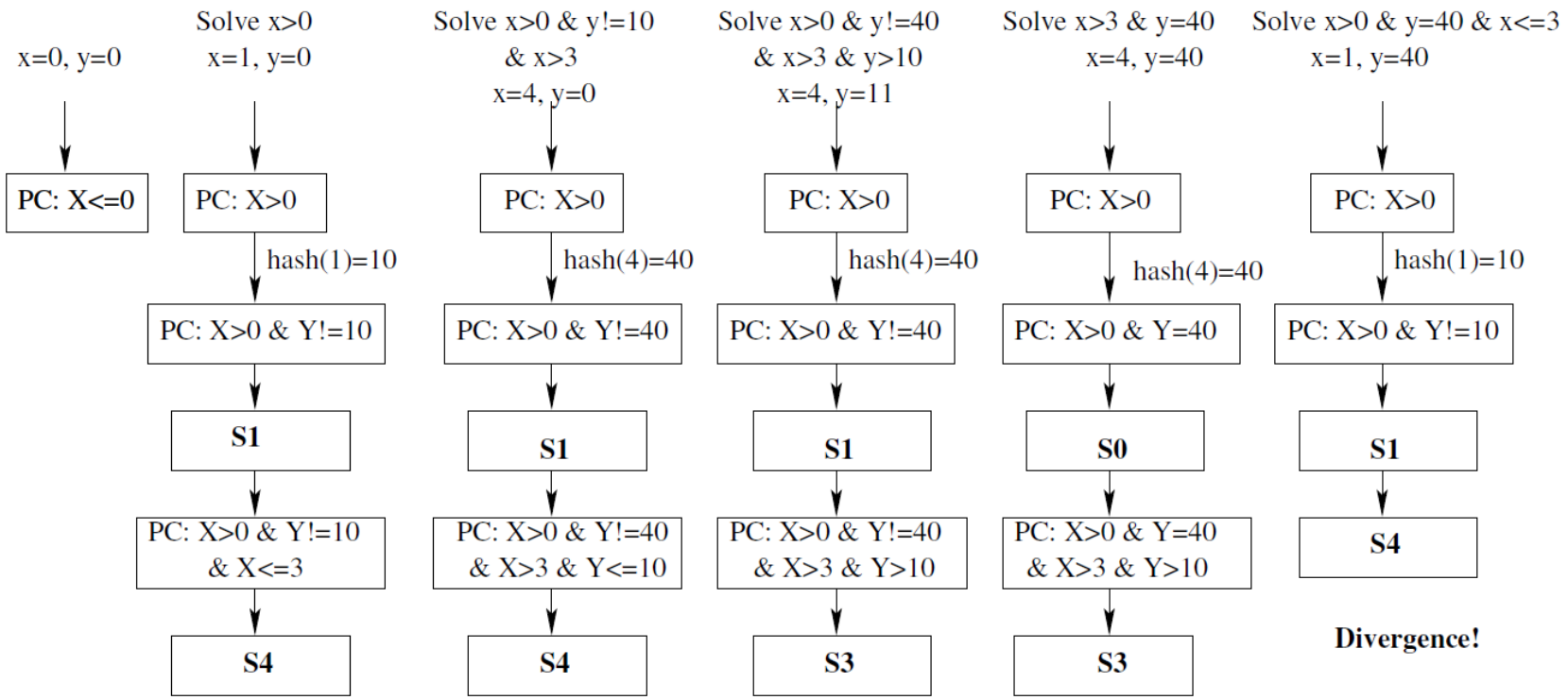
see:
Symbolic Execution with Mixed Concrete-Symbolic Solving
Pasareanu, Rungta, Visser.
ISSTA 2011

EXE



EXE: Automatically Generating Inputs of Death
CRISTIAN CADAR, VIJAY GANESH, PETER M. PAWLOWSKI, DAVID L. DILL and DAWSON R. ENGLER
Stanford University
CCS 2006

DART (Directed Automated Random Testing)



Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. (PLDI '05)

KLEE

KLEE

- Cristian Cadar, Daniel Dunbar, Dawson Engler, *Stanford University*
- *KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs.* OSDI'08
- KLEE ist der Nachfolger von EXE
- aus der Erfahrung mit EXE gelernt:
  - Constraint Solving Optimierungen
  - kompakte Darstellung von Programmzuständen
  - Such-Heuristiken um hohe Code Coverage zu erreichen
  - einfacher Ansatz zur Behandlung der externen Umgebung

KLEE Features

- arbeitet auf LLVM Code (z.B. von GNU-C erzeugt) statt auf Programmiersprache
- erlaubt die Auszeichnung von Variablen als `symbolic`
- Auswahl des nächsten auszuführenden Zustands durch verzahnte zufällige Pfadauswahl und *Coverage-Optimized Search* = wähle den Pfad, der am wahrscheinlichsten neue Code überdeckt
- erreicht eine durchschnittliche Code-Abdeckung von 90%
- einfaches Umgebungsmodell: Systemcalls emulieren in einer Bibliothek ein vereinfachtes Modell des Betriebssystems
- erlaubt symbolische Eingabewerte
- erzeugt (konkrete) Testfälle für alle Pfade
- erlaubt den Re-Run von Testfällen ohne KLEE-Instrumentierung essentiell, da False Positives möglich!

Klee Fehlersuche

- „gefährliche“ Operationen
  - Assertions
  - Bereichsgrenzen
  - Buffer Overflow
  - Pointer Dereferéncing
- Crosscheck äquivalenter Programme

KLEE: Fallstudien

- 89 Programme der GNU Coreutils
  (V6.1: 80.000 LoC Bibliothek, 61.000 LoC Utilties)
  "The single most well-tested suite of open-source applications"
  - 1995: GNU Coreutils haben deutlich weniger Defekte als kommerzielle Unix Systeme
  - Letzte Schwachstelle 2005 entdeckt
  - KLEE spürte 10 fatale Fehler auf
- andere Fallstudien:
  - Busybox
  - Minix
  - HiStar OS
- Dawson Engler hat eine Firma für Static Analysis mitgegründet: Coverity

```
paste -d\\ abcdefghijklmnopqrstuvwxyz
pr -e t2.txt
tac -r t3.txt t3.txt
mkdir -Z a b
mkfifo -Z a b
mknod -Z a b p
md5sum -c t1.txt
ptx -F\\ abcdefghijklmnopqrstuvwxyz
ptx x t4.txt
seq -f %0 1
```

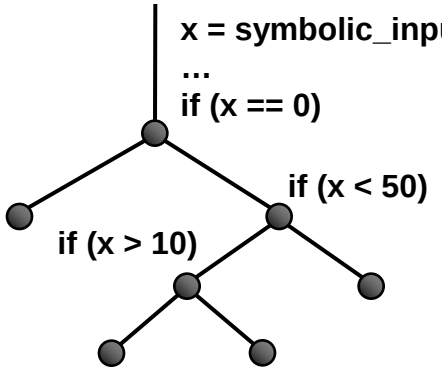
t1.txt: "\t \tMD5("
t2.txt: "\b\b\b\b\b\b\t"
t3.txt: "\n"
t4.txt: "a"

KLEENET

Protokollverifikation

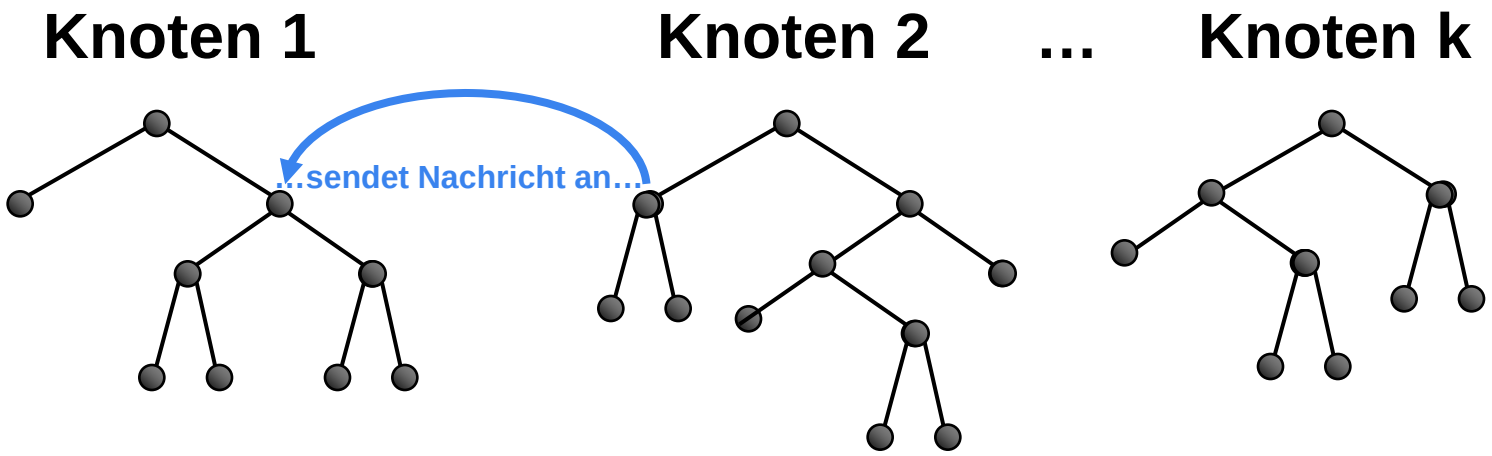
- Raimondas Sasnauskas, Communication and Distributed Systems, RWTH Aachen University + Carsten Weise, Embedded Software Laboratory, RWTH Aachen im Rahmen des UMIC Excellence Clusters
- Eigentliches Ziel: Modelchecking existierender Protokoll Implementierungen
- Nach vielen gescheiterten Versuchen: KLEE konnte unsere Protokoll-Software verarbeiten
- Aber: KLEE beschränkt sich auf Single-Processor Systeme – keine Nebenläufigkeit
- Wie kann man KLEE auf verteilte Systeme erweitern?
- Antwort: KleeNet

Von einem zu vielen Knoten

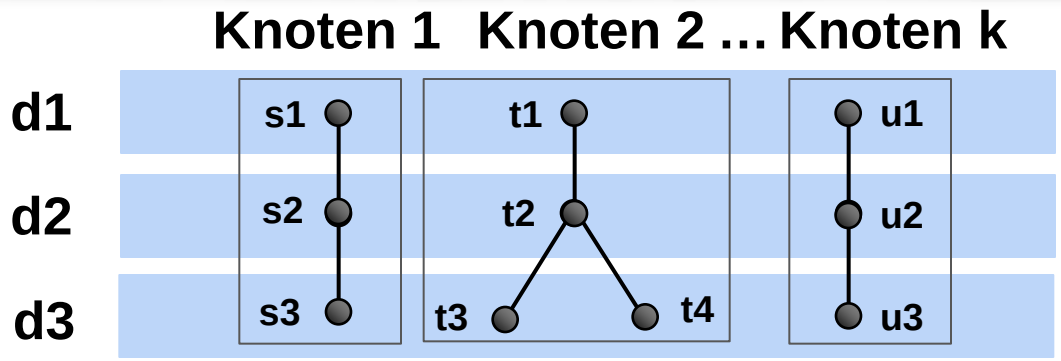


Path 1: { x = 0 }
Path 2: { x > 10 ∧ x < 50 }
Path 3: { x ≥ 50 }
Path 4: { x ≠ 0 ∧ x ≤ 10 }

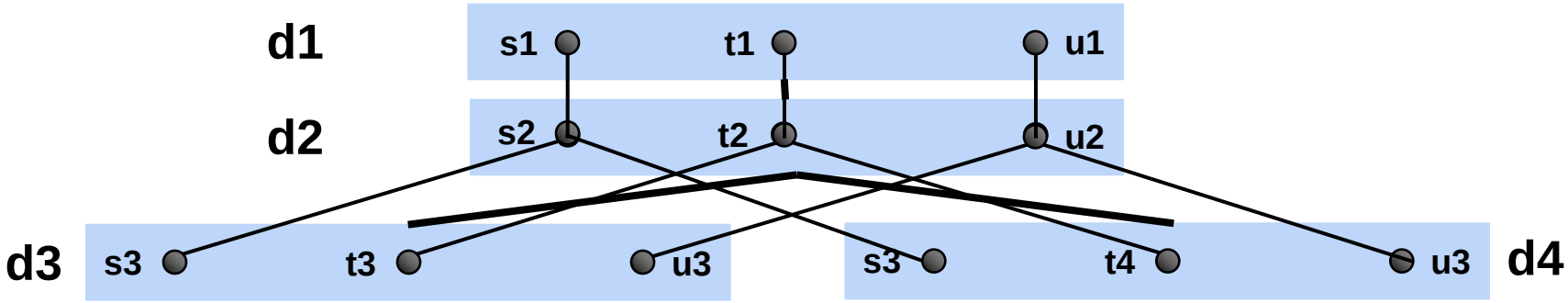
Testcase 1: x = 0
Testcase 2: x = 42
Testcase 3: x = 314
Testcase 4: x = -7



Naive Simulation



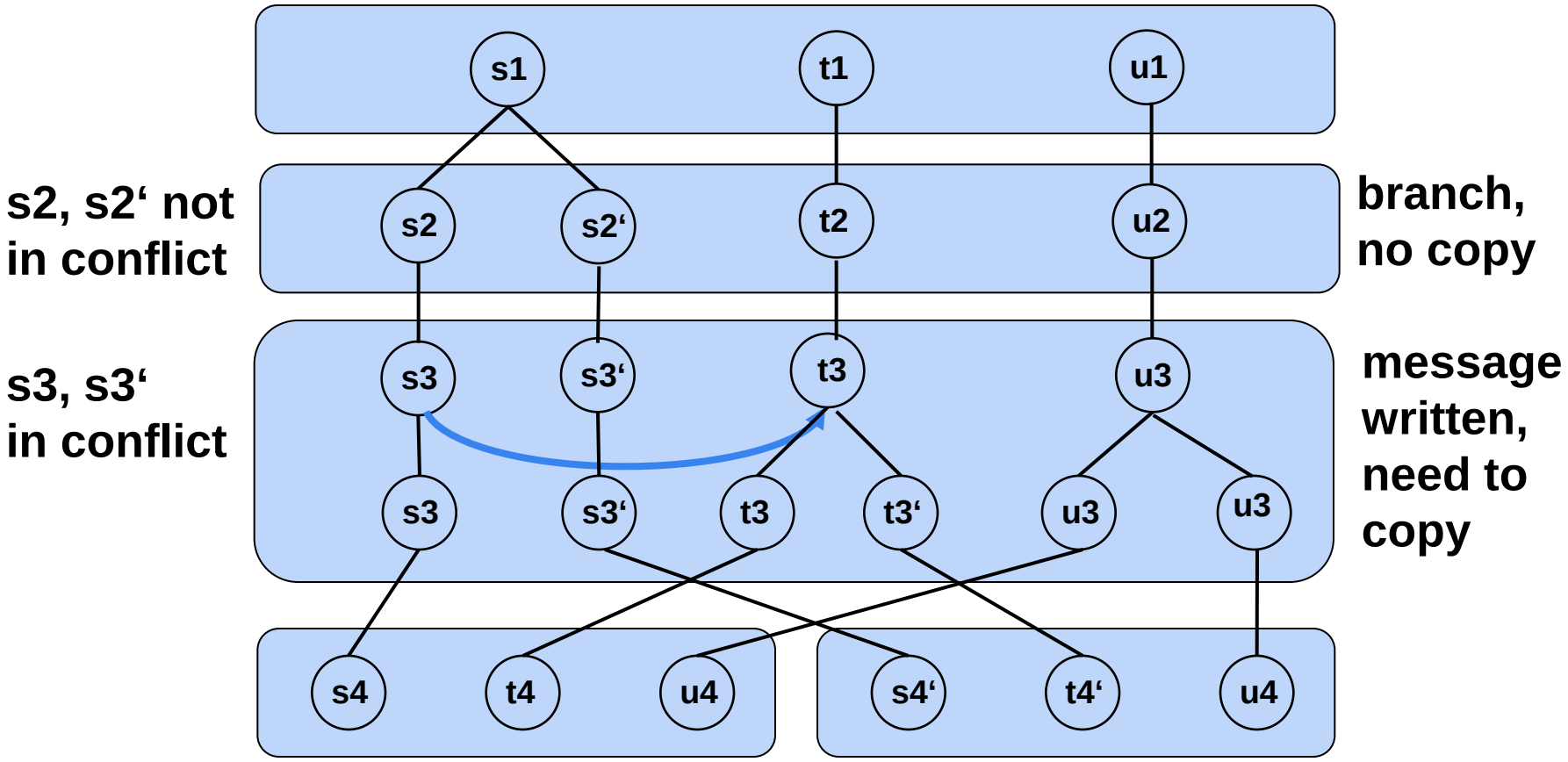
verteilter Zustand



COB – Copy on Branch

- Die naive Simulation führt (wie bekannt) zu einer Zustandsexplosion
- Es werden viele identische Teilzustände der Knoten erzeugt
- Die Zustandsexplosion entsteht durch das Kopieren von Zuständen
- Wir haben Verbesserungen des Kopier-Verhaltens untersucht
- Die naive Simulation nennen wir „Copy On Branch“ (COB)–
  wenn ein Prozess verzweigt, müssen alle Zustände kopiert werden
- Wir haben zwei Verbesserungen implementiert
  - Copy On Write (COW)
  - Super Distributed States (SDS)

COW: Copy on Write

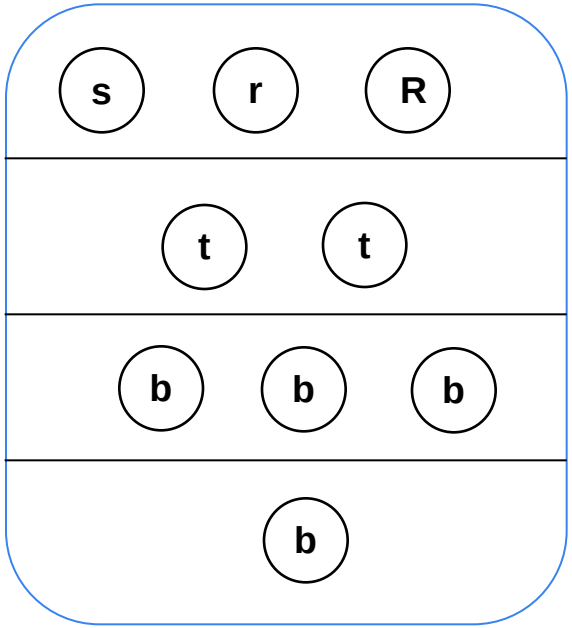


conflict: message history ist verschieden

Super Distributed States

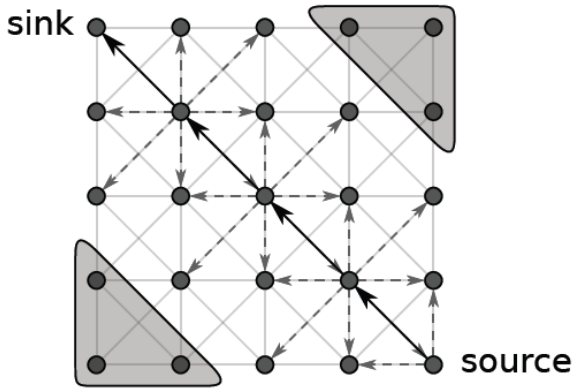
- War das Duplizieren von u3 nötig?
- Vielleicht – das hängt davon ab, ob sich die Kommunikation zwischen s und t später ursächlich auf u auswirkt
- Offensichtlich: Könnte wir den Kommunikationsgraphen partitionieren, so müssten wir nur in der jeweiligen Partition duplizieren, aber nicht die anderen Zustände
- Diese Überlegungen führen zu weiteren Strategien zum Unterdrücken von unnötigem Duplizieren

s: sender
t: target
r: rival
R: super-rival
b: bystander



Performance-Vergleich

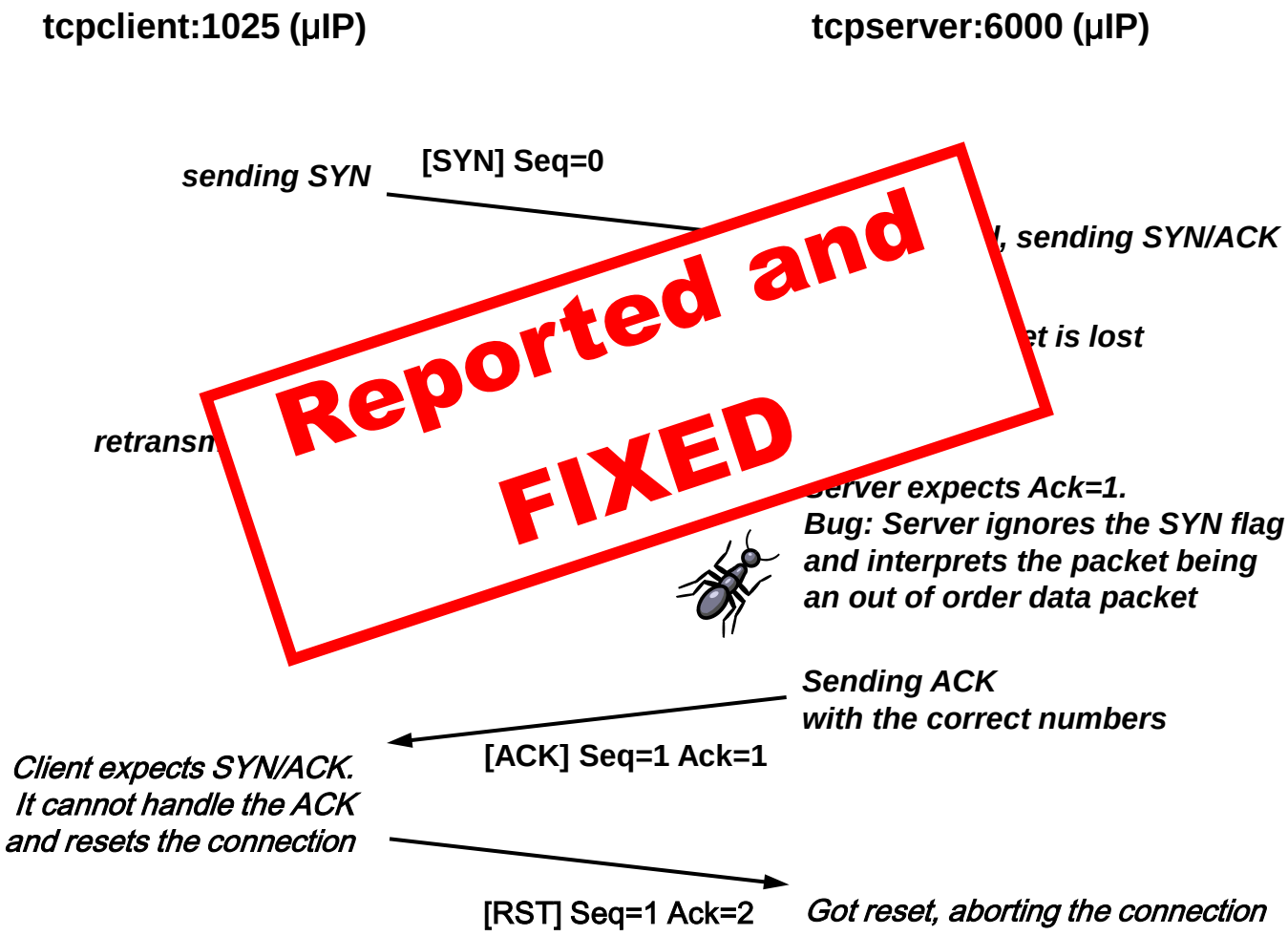
- 25, 49, und100 Contiki Knoten über Rime Communication Stack



State mapping algorithm	Runtime	States	RAM
Copy On Branch (COB)	9h:39m (**aborted**)	1,025,700	38.1 GB
Copy On Write (COW)	1h:38m	30,464	3.4 GB
Super DStates (SDS)	19m	4,159	1.6 GB

TABLE I
TEST RESULTS FOR THE 100 NODE SCENARIO WITH SYMBOLIC PACKET DROPS ENABLED.

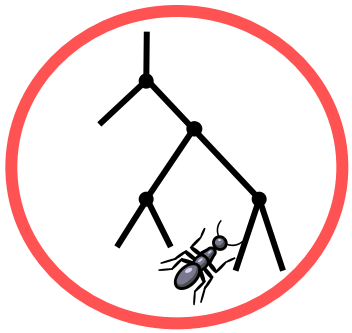
Fallstudie: µIP Stack



KleeNet Project

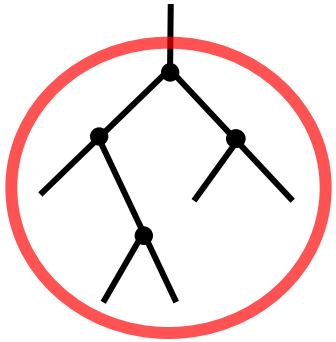
- High-Coverage Sensornet Testing Before Deployment
  - Employs symbolic distributed execution of unmodified software

Packet with “any” data



KleeNet

- **Automatically explores all possible execution paths**
- **Detects corner-case interaction bugs**
- **Generates concrete input for bug replay**
- **Found bugs in widely deployed sensornet software**



  - Developed and formalized symbolic distributed execution
  - Integrated KleeNet into a widely used sensornet simulator

KleeNet Abschließendes

- Raimondas Sasnauskas, Oscar Soria Dustmann, Benjamin Lucien Kaminski, Carsten Weise, Stefan Kowalewski and Klaus Wehrle. *Scalable Symbolic Execution of Distributed Systems.* ICDCS
- Raimondas Sasnauskas, Olaf Landsiedel, Muhammad Hamad Alizai, Carsten Weise, Stefan Kowalewski and Klaus Wehrle. *KleeNet: Discovering Insidious Interaction Bugs in Wireless Sensor Networks Before Deployment*, IPSN 2010
- Oscar Soria Dustmann, Raimondas Sasnauskas and Klaus Wehrle. *Symbolic System Time in Distributed Systems Testing.* TAICPART'2012 April 2012
- Download: http://www.comsys.rwth-aachen.de/research/projects/kleenet/

WEITERES ZUM TESTEN

Testen des Lebenzyklus mobiler Anwendungen

- Dominik Franke, Lehrstuhl für Informatik 11, RWTH Aachen
- Franke, D., Kowalewski, S., Weise, C., and Prakobkosol, N., *Testing Conformance of Lifecycle-Dependent Properties of Mobile Applications*. ICST 2012.
- http://embedded.rwth-aachen.de/doku.php?id=lehrstuhl:mitarbeiter:franke

Testfallgenerierung aus stochastichen Modellen

- Timed Statistical Modelchecker: http://people.cs.aau.dk/~adavid/smc/
- *Time for Statistical Model Checking of Real-time Systems*. Alexandre David, Kim Guldstrand Larsen, Axel Legay, Marius Mikucionis, Zheng Wang. CAV 2011

Off-line Testing

- Verschiedene Ansätze auf der ICTSS 2012
- Interessanter Ansatz für die Praxis: Analyse von Logfiles als Test
- H. Barringer, A. Groce, K. Havelund, and M. Smith. *Formal analysis of log files*. Journal of Aerospace Computing, Information, and Communication, 7(11) 2010.

Modellgenerierung durch Trace-Analyse

- Durch Testen zum Modell
- Basiert auf dem L* Algorithmus (Angluins Algorithmus) zum Lernen von Automaten
- F. Aarts, H. Kuppens, G.J. Tretmans, F.W. Vaandrager, and S. Verwer, *Learning and Testing the Bounded Retransmission Protocol*. ICGI 2012

Test-Tagungen 2013

- ICST 2013 in Luxemburg (http://www.icst.lu/)
  - 18.-22.März
  - A-MOST Workshop
  - TAICPART Workshop
- ICTSS 2013 in Istanbul (http://ictss.sabanciuniv.edu/)
  - 13.-15.November

Schluß

Danke für die Aufmerksamkeit