Karlsruhe Institute of Technology

Automatic Heavy-weight Static Analysis Tools for Finding Bugs in Safety-critical Embedded C/C++ Code

David Faragó, Florian Merz, Carsten Sinz

Juni 2014

Institute for Theoretical Computer Science
Static Analysis Spinoff LLBMC

Problem am Beispiel Automotive

Steigende Software-Komplexität:
80 - 90 Controller pro Auto
bis zu 10 Millionen Zeilen Programmcode

Kürzere Time to Market:
Reduziert von 60 Monaten
auf heute 18-24 Monate

Mehr Rückrufe:
2009 bis 2012 wurden mehr Autos zurückgerufen als verkauft,
2013 war jeder zweite Rückruf softwarebedingt

Problem am Beispiel kardiologischer Geräte

Steigende Anzahl an kardiologischen Geräten:
2008 ca. 500.000 Herzschrittmacher und Defibrilatoren verkauft
Jährlich über 8% Wachstum

Mehr Rückrufe und Unfälle (in USA laut MAUDE und FDA):
Letzten 10 Jahre: Rückrufe mehr als verdoppelt
2013 über 2000 Rückrufaktionen (ca. 10% lebensgefährliche)
und ca. 11.000 Todesfälle

Steigende Softwarekomplexität:
Ca. 100.000 Zeilen Programmcode pro Gerät
Anteil an software-bedingten Rückrufen steigt ständig
2013 war jeder zweite Rückruf softwarebedingt

Safety-critical QA: aktueller Stand

- Klassisches Testen: Randfälle werden nicht entdeckt

➠ zusätzlich light-weight statische Analyse: Randfälle werden nicht entdeckt

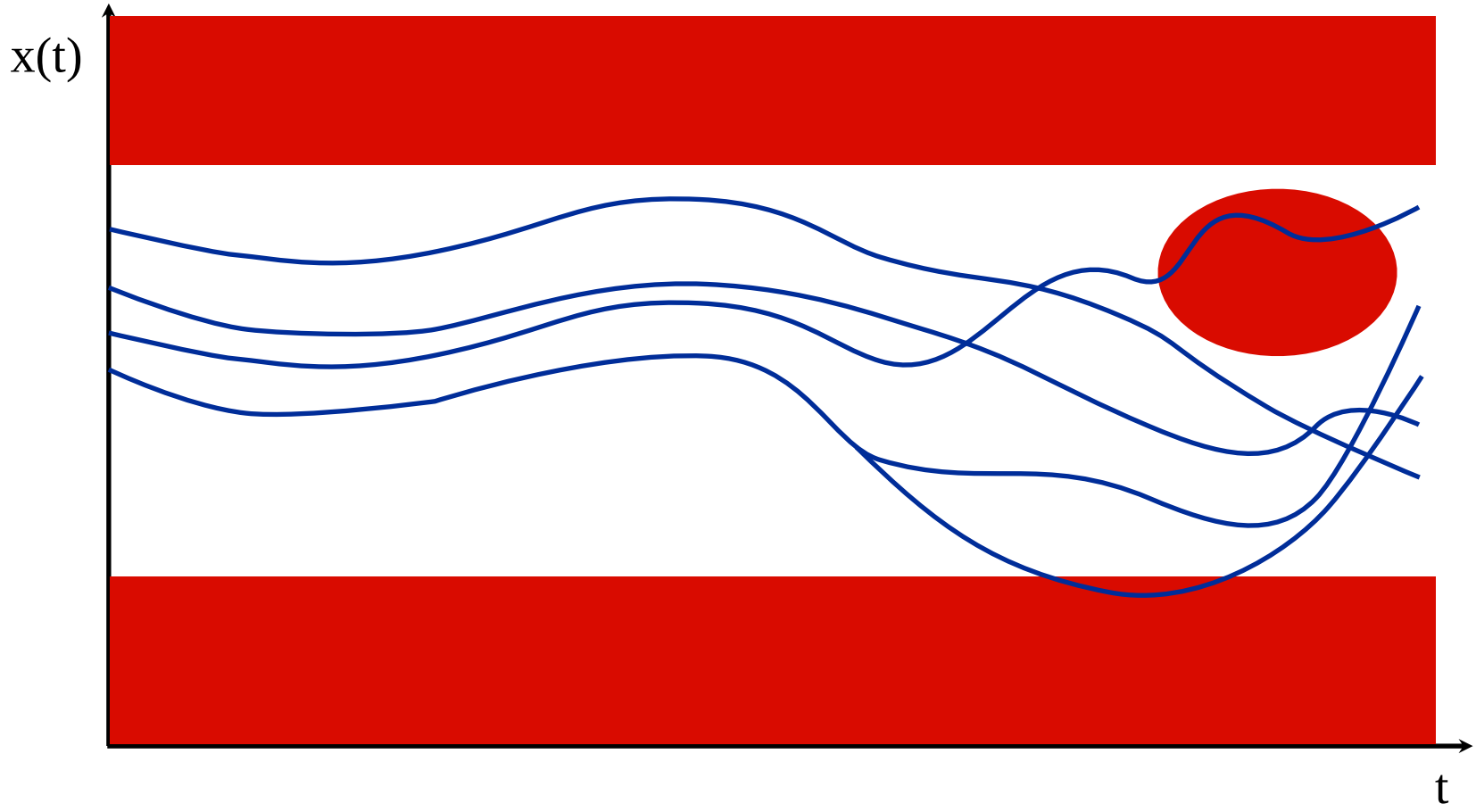
➠ lieber heavy-weight statische Analyse

- kommerzielle heavy-weight statische Analyse-Werkzeuge
  - Coverity, Polyspace, Astrée
  - beruhen auf der Technik „Abstrakte Interpretation“:
    Abstrahiert von Ausführungspfaden zu Klassen von Ausführungspfaden, z.B. mittels Intervallen von Variablenwerten

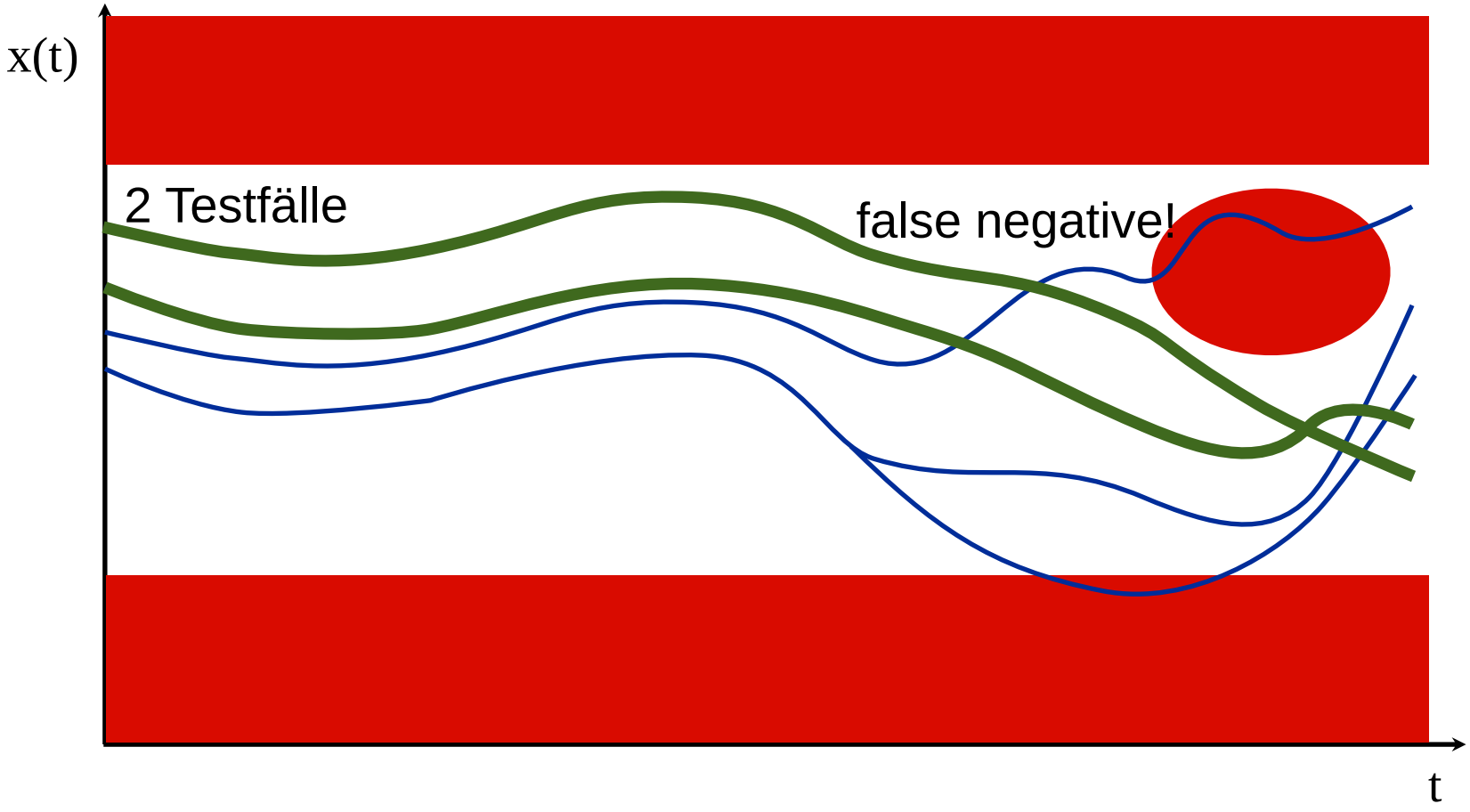
  ➠ mangelnde Präzision (false positives oder false negatives) und
  wenig Unterstützung bei Fehlerbehebung (keine „Fehlertraces“)

? Safety-critical QA? Welche Tools? Wieso? Zufrieden?

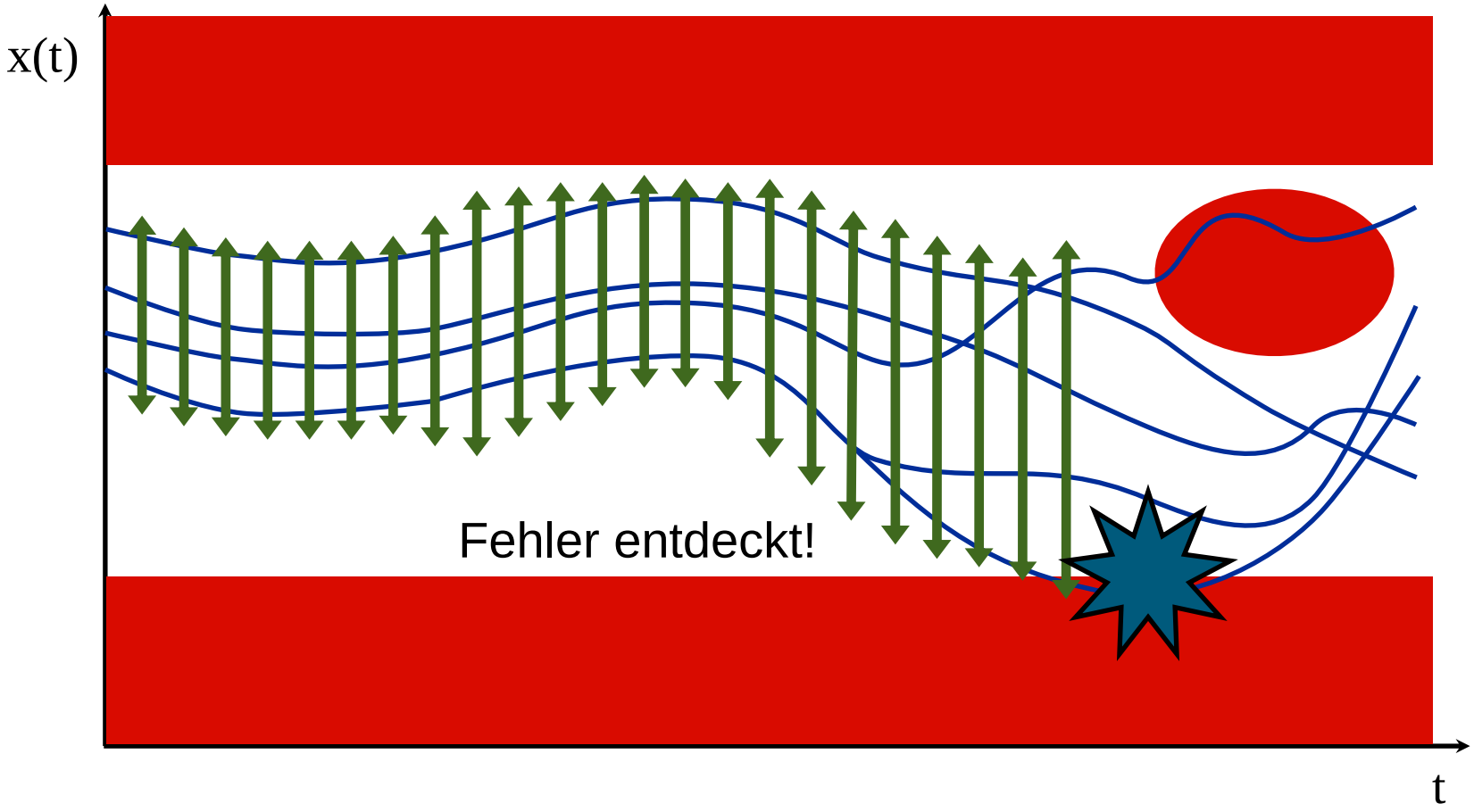
Visualisierung des Programmverhaltens



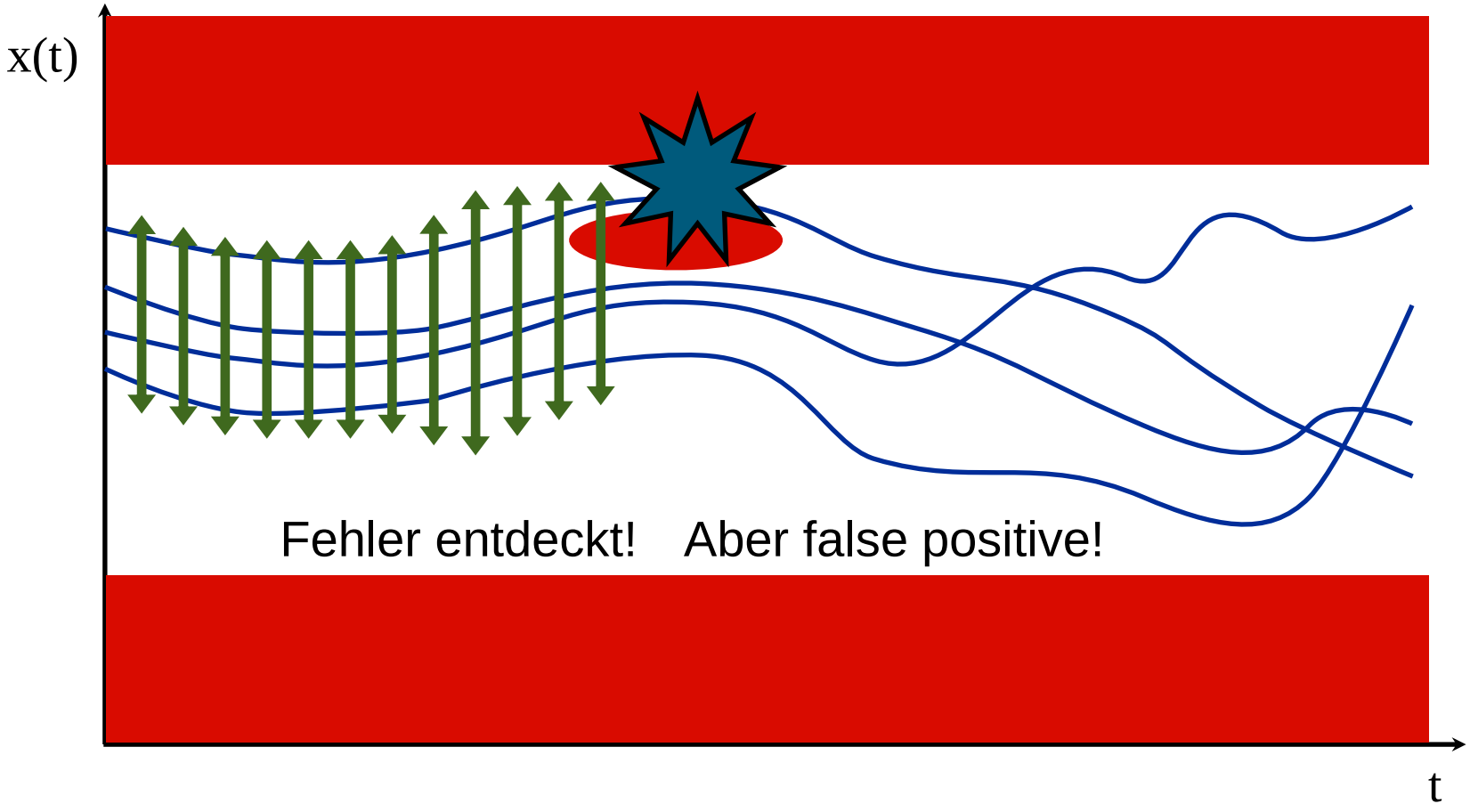
Visualisierung Testing



Visualisierung Abstrakte Interpretation



Visualisierung Abstrakte Interpretation 2



Software Bounded Model Checking mit LLBMC

- Analysiert **alle Inputs** eines Programms und **alle Pfade** bis zu einer **festen Länge *b***

➠ **Feste Länge *b*:** bis zum Limit Schleifen abrollen und Funktionen inlinen

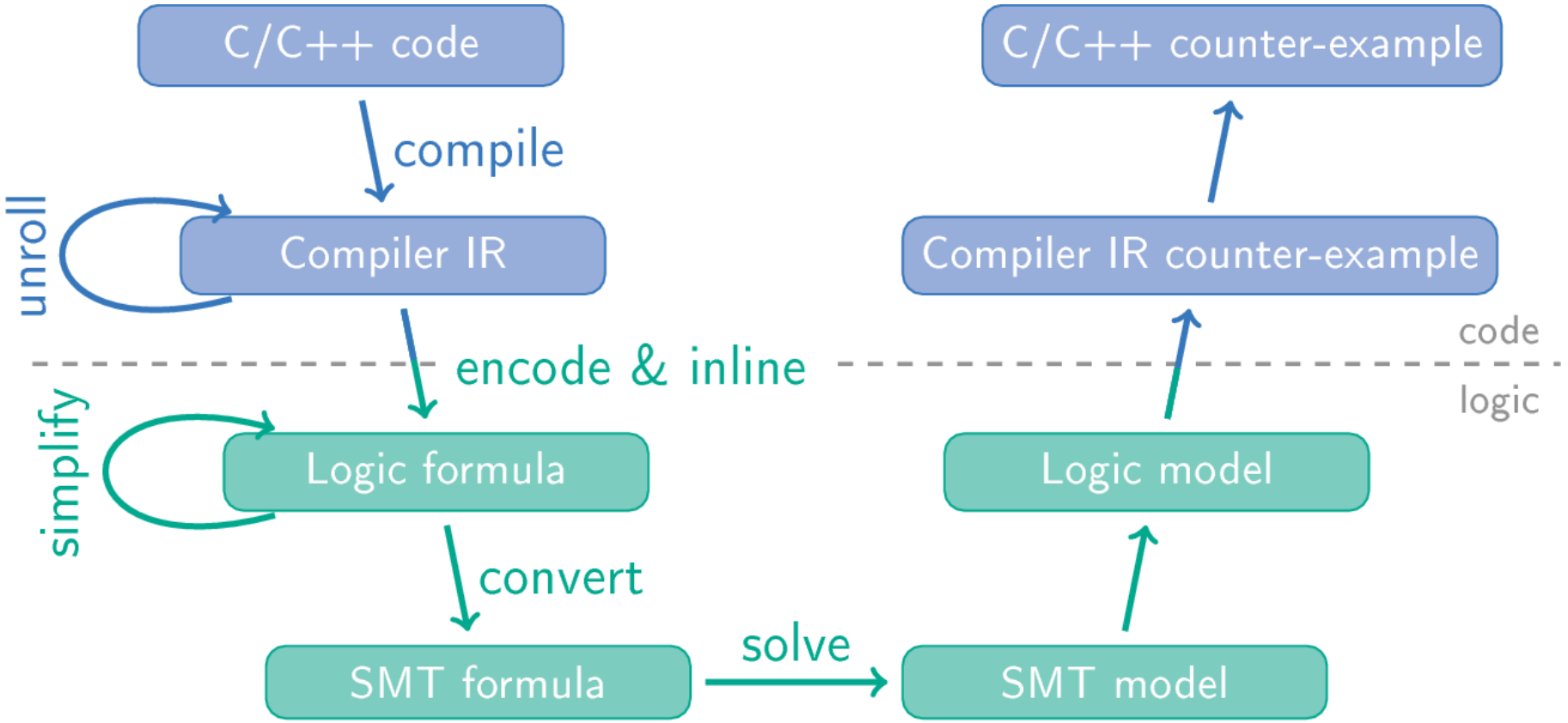
➠ Folgende **Fehlerklassen** können vollautomatisch, entscheidbar überprüft werden

- arithmetische Fehler (over-/under-flow, division by zero)
- Bit-Operator Fehler (invalid bit shift)
- Speicherfehler (array index out of bound, buffer overflow, illegal pointer access/pointer dereference, memory leaks, invalid/double free, use after free)
- Benutzerdefinierte Eigenschaften (Assertions)

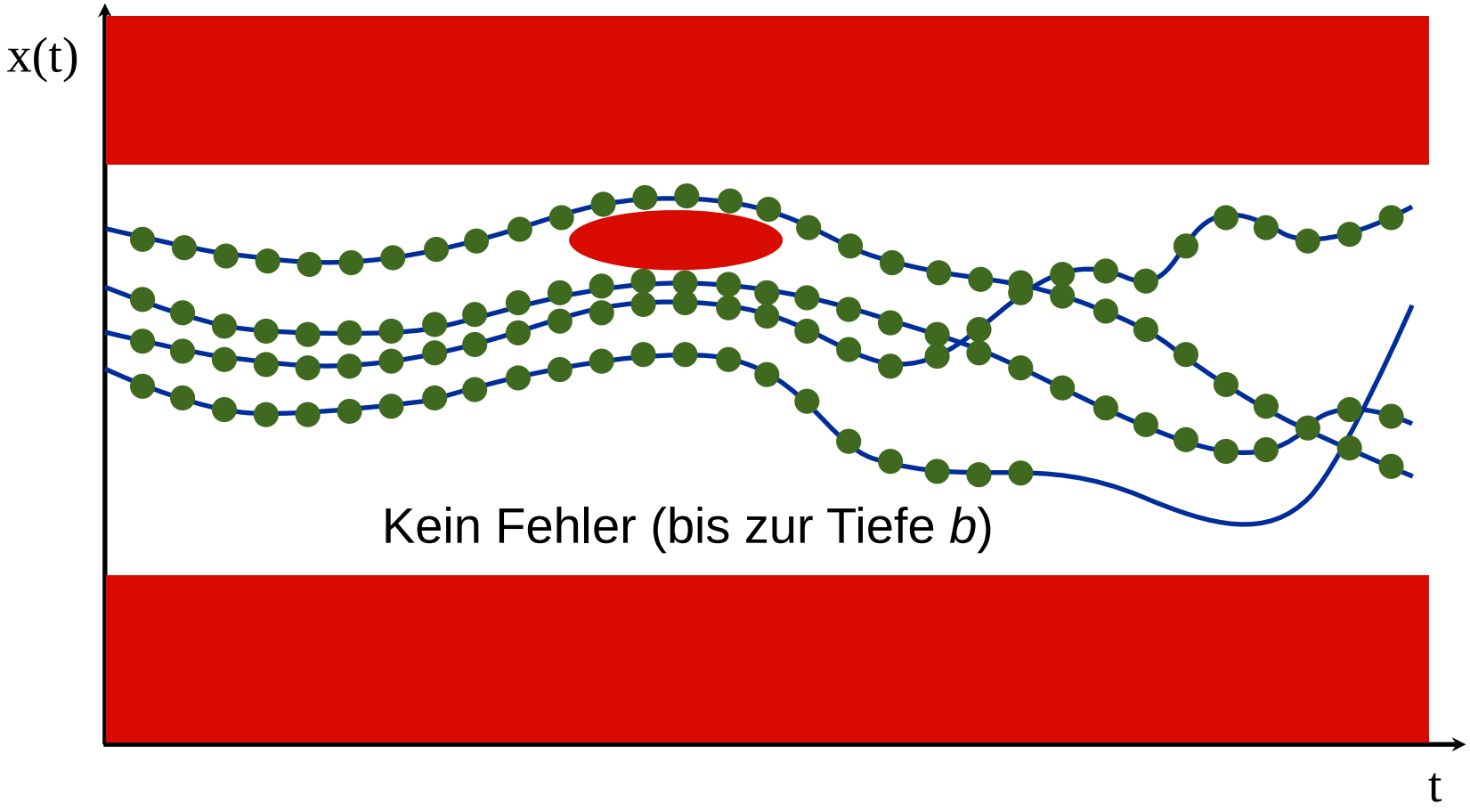
- Programm mit fester Länge wird in eine **logische Formel** übersetzt (QF_BVA)
- Semantik auf bit-Ebene bleibt erhalten
- Überprüfe Formel mittels SAT/SMT-Solver
- Übersetze ggf. Gegenbeispiel zurück in Fehlertrace auf Programm-Ebene

- Herausforderung: **Skalierbarkeit**

Technik von LLBMC



Visualisierung LLBMC



Wettbewerb: Details

Eigen-schaft	Klassisch Testen	MBT	Polyspace und Astrée	Coverity	LLBMC
Basis-Technologie	Testframe-work	automatische Testfall-Generierung	Abstrakte Interpretation	Abstrakte Interpretation	Compiler-Technik; SMT-Solver
Präzision	schlecht; sehr niedrige Coverage	gute Coverage (MC/DC und strengere) schwierig	bis zu 50% der Warnungen sind false positives	ca. 15% false posi-tives, evtl. viele false negatives	innerhalb *b* keine false negatives & false positives
Klarheit Analyse-ergebnis	unklar wieviel false negatives	unklar wieviel false negatives (aber gute Coverage limitiert)	potentielle false positives sind gelb markiert; manu-elle Nacharbeit	unklar wieviel false positives & false negatives	tritt praktisch nicht auf
Darstellung Analyse-ergebnisse	detaillierter Fehlertrace; Testfall	detaillierter Fehler-trace; Testfall; abstrakter Testfall	Intervalle für mögliche Werte	partiell Datenfluss	detaillierter Fehlertrace
Inkremen-telles Vorgehen	Tests ändern sich häufig; refactoring	Modell ändert sich häufig	jede Änderung erfordert manuelle Nacharbeit	Beschleunigung durch Impact-Analysis	Impact-Analysis geplant; keine ma-nuelle Nacharbeit

Wettbewerb: Übersicht

Eigenschaft	Klassisch Testen	MBT	Polyspace & Astrée	Coverity	LLBMC
Konfigurations-aufwand	●	◑	○	●	●
Präzision	○	◑	◑	◑	●
Klarheit Analyseergebnis	○	◑	◑	○	●
Darstellung Analyseergebnis	●	●	○	○	●
Inkrementelles Vorgehen	◑	●	○	◑	◑

○ schlecht ◑ mittel ● gut

Status Quo

Partner und Kundenkontakte

DAIMLER

HELMHOLTZ ASSOCIATION

Gewonnene Medaillen der International Software Verification Competitions 2012-2014

2014 — **2015**

Seit ’08
Forschung

Apr ’14 – Jun ’14
Konzeptionsphase

Jun ’14 – Okt ’14
1. Validierung

Ab Nov ’14
weitere Validierungsphasen

Seit Mai ’14
Entwicklung GUI und Webapplikation

Vielen Dank für eure Aufmerksamkeit

- **Kontakt:**
  Mail farago@kit.edu
  Fon +49 721 608-47322
  Fax +49 721 608-44211
  Web http://llbmc.org

- **Büro:**
  David Faragó
  Institut für Theoretische Informatik
  Gebäude 50.34
  Am Fasanengarten 5
  76131 Karlsruhe

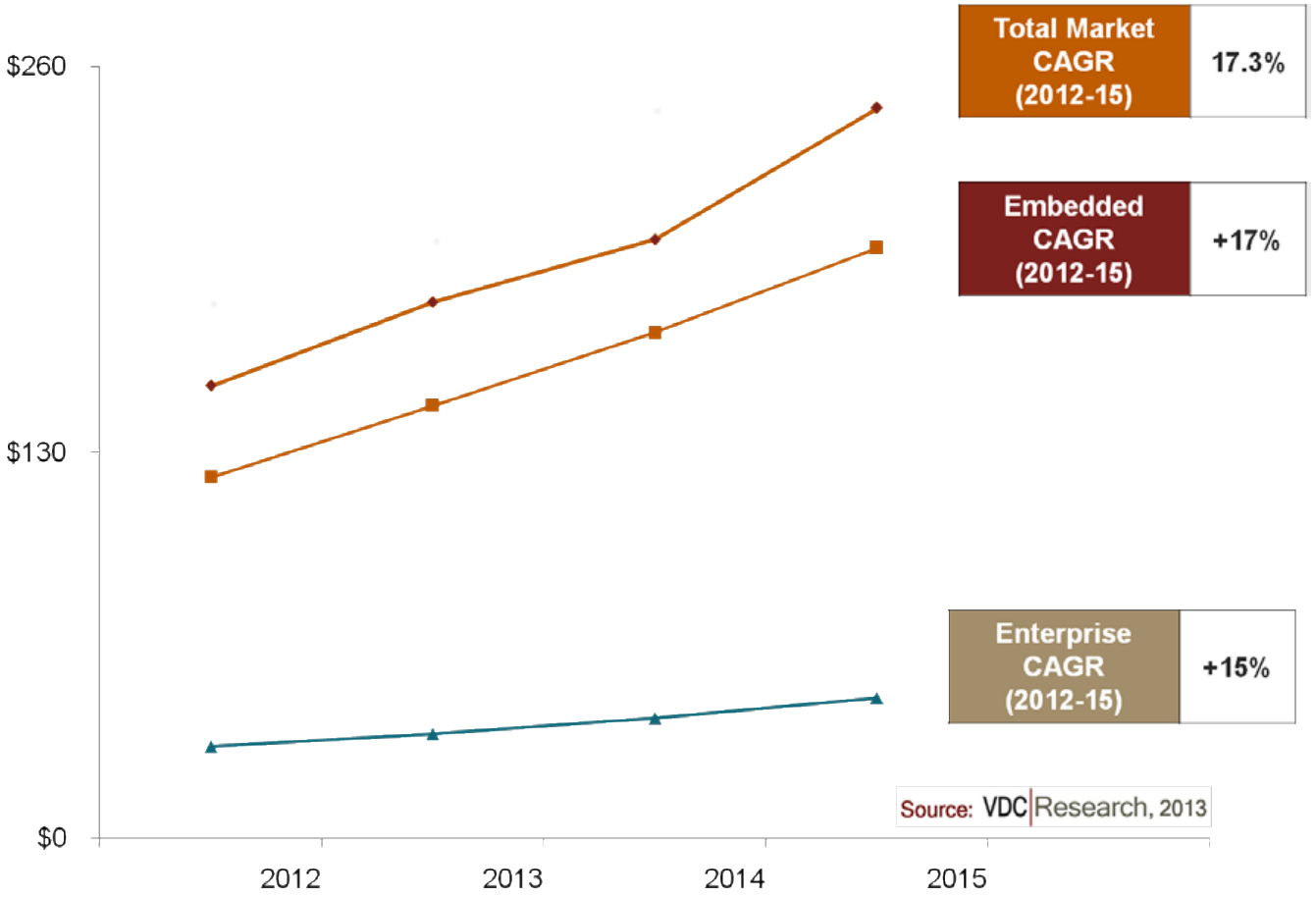
Besuchen Sie unseren Stand auf der

24. bis 26. Februar 2015, Nürnberg
Ansprechpartner: Dr. Carsten Sinz

Karlsruhe Institute of Technology

Ergänzendes Material

Marktanalyse Statische Analyse-Werkzeuge



Global Static Analysis Tools Growth Trends 2012-2015 (US $M)
[VDC Research Report, 28.10.2013]

Geschäftsmodell

- Marktvolumen: ca. 400-500 Mio. in der zweiten Hälfte des Jahrzehnts
  [Schätzung basierend auf Forbes-Artikel vom 20.02.2013: Coverity Growing at 30% And Poised for IPO]
- Mittelfristig angestrebter Weltmarktanteil: > 5%
- First-Mover im Bereich Software Bounded Model Checking
- Market-Pull durch steigender Aufwand der QA und der Anzahl an Rückrufen
- Flexible Vermarktungsstrategie, Nutzung in der Cloud oder On-Premises
- Anwendungsgarantie: Consulting stellt sicher, dass die Anwendung optimal genutzt und die Ergebnisse richtig interpretiert werden

Vorteile von LLBMC

- Softwarefehler besser erkennen: keine false positives/false negatives
- Ursachen klar veranschaulichen: mittels Fehlertrace
- Schnell höchste Softwarequalität garantieren
- Time-to-Market Vorgaben sichern
- Hohe Kosten vermeiden, z.B. Rückrufkosten

Speichermodell

$$\texttt{valid\_mem\_access}(m,p,s) \equiv \bigvee_{\substack{m' \preceq m \\ I:\, m' = \texttt{malloc}(\tilde{m},q,t)}} \Big(c_{\texttt{exec}}(I) \wedge q \neq \texttt{NULL} \wedge (q \leq p \leq q+t-s) \wedge \neg\texttt{deallocated}(m',m,q) \Big)$$

$$\texttt{deallocated}(m,m',p) \equiv \bigvee_{\substack{m \preceq m^* \preceq m' \\ I:\, m^* = \texttt{free}(\tilde{m}^*,q)}} \Big(c_{\texttt{exec}}(I) \wedge p = q \Big)$$

$$\texttt{non\_overlap}(p,s,q,t) \equiv (p+s \leq q) \vee (q+t \leq p)$$

$$\texttt{malloc\_assumption}(m,p,s) \equiv p = \texttt{NULL} \vee \Big[(p \geq \texttt{MEM\_MIN} \wedge p+s-1 \leq \texttt{MEM\_MAX}) \wedge \bigwedge_{\substack{m' \preceq m \\ I:\, m' = \texttt{malloc}(\tilde{m},q,t)}} \Big(c_{\texttt{exec}}(I) \wedge q \neq \texttt{NULL} \wedge \neg\texttt{deallocated}(m',m,q) \implies \texttt{non\_overlap}(p,s,q,t) \Big) \Big]$$

$$\texttt{valid\_free}(m,p) \equiv p = \texttt{NULL} \vee \bigvee_{\substack{m' \preceq m \\ I:\, m' = \texttt{malloc}(\tilde{m},q,t)}} \Big(c_{\texttt{exec}}(I) \wedge p = q \wedge \neg\texttt{deallocated}(m',m,q) \Big)$$

$$\texttt{no\_memory\_leaks}(m) \equiv \bigwedge_{\substack{m' \preceq m \\ I:\, m' = \texttt{malloc}(\tilde{m},p,s)}} \Big[c_{\texttt{exec}}(I) \implies \bigvee_{\substack{m' \preceq m^* \preceq m \\ J:\, m^* = \texttt{free}(\tilde{m}^*,q)}} \Big(c_{\texttt{exec}}(J) \wedge p = q \Big) \Big]$$