Outline

Lecture 1, part 1
• Motivation
• Model Checking

Lecture 1, part 2
• State/Event-based software model checking

Lecture 2
• Component Substitutability
Substitutability Check

Assembly \( \mathcal{A} \)

Component \( C \)

Component \( C' \)
Motivation

- Component-based Software
  - Software modules shipped by separate developers
  - An assembly consists of several components
  - Undergo several updates/bug-fixes during their lifecycle

- Component assembly verification
  - Necessary on update of any component
  - High verification costs of global properties
  - Instead check for substitutability of new component
Substitutability Check

• Given old and new components $C$, $C'$ and assembly $\mathcal{A}$
  – Check if $C'$ is substitutable for $C$ in $\mathcal{A}$

• Two phases:
  – Containment check
    • All behaviors of the previous component contained in new one
  – Compatibility check
    • Safety with respect to other components in assembly: all global specifications satisfied

• Formulation
  – Obtain a finite behavioral model of all components by abstraction: labeled kripke structures
  – Use regular language inference in combination with a model checker
Predicate Abstraction into LKS

• Labeled Kripke Structures
  – \(<Q, \Sigma, T, P, \mathcal{L}>\)

• Composition semantics
  – Synchronize on shared actions

• Represents abstractions

• State-event traces
  – \(<p, \beta, q, \gamma, \ldots, \ldots>\)
Component Interface LKS

• Component
  – A set of communicating concurrent C programs or libraries
    • No recursion, inlined procedures
  – Abstracted into a Component Interface LKS
  – Communication between components is abstracted into interface actions
Learning Regular languages: L*

- Forms the basis of *containment* and *compatibility* checks
- L* proposed by D. Angluin
- Polynomial in the number of states and length of counterexample

L* learner

Minimally adequate Teacher

Modelchecker

Unknown Regular Language

minimum DFA

IsMember( trace ρ )

IsCandidate( DFA D )

±Counterexample/ Yes
Containment

- **Goal:**
  - Learn *useful* behaviors from previous component into the new one

- **Given Component LKSs:** $M$, $M'$ and Bug LKS $B$
  - Unknown $U = L(M') \cup (L(M) \setminus L(B))$
  - Iteratively learn the DFA $SM'_i$ using $L^*$

- **Model checker**
  - IsMember Query: $\rho \in U$
  - IsCandidate Query: $U \equiv L(SM'_i)$
Containment (contd.)

IsMember Query: $\rho \in U$

IsCandidate Query: $U \equiv L(SM'_i)$
Containment (contd.)

- In contrast to known \textit{Refinement}-based approaches
  - Containment allows adding \textit{new} behaviors in $M'$
    e.g. $M$, $M'$ have different interleavings of same interface actions
  - Erroneous new behavior detected in Compatibility check

- Finally $SM'$
  - substitutable candidate
  - may not be safe with respect to other components
  - must verify the global behavioral specifications
Compatibility check

- Assume-guarantee to verify assembly properties

\[ \begin{align*}
R_1: & \quad M_1 \parallel A \models P \\
R_2: & \quad M_2 \models A \\
\hline
M_1 \parallel M_2 \models P
\end{align*} \]

- Generate a (smaller) environment assumption **A**
  - **A**: most general environment so that **P** holds
  - Constructed iteratively using L* and \( R_1, R_2 \)
Compatibility check

\[ R_1: \quad M_1 \parallel A_i \vdash P \]

\[ R_2: \quad M_2 \models A_i \]

-CE for \( A_i \)

CE Analysis

True CE \( M_1 \parallel \rho \neq P \)

False CE

\[ +CE \text{ for } A_i \]

\[ \rho \]

\[ true \]

\[ true \]

\[ true \]

L* Assumption Generation

\( A_i \)
Compatibility check (contd.)

• Generate a most general assumption for SM’
  – $M_1 = SM’$
  – $M_2 = \mathcal{M}\setminus M$ (all other component LKSs)

• Membership queries:
  – $SM’ \parallel \rho \subseteq P$

• Candidate queries:
  – $SM’ \parallel A \subseteq P$
  – $M_2 \subseteq A$

• CE analysis: $SM’ \parallel CE \subseteq P$
  – Yes $\Rightarrow$ False CE
  – No $\Rightarrow$ True CE
CEGAR

• Compatibility check infers
  – Either SM’ is substitutable
  – Or counterexample CE

• CE may be spurious wrt C, C’
  – CE is present in component LKS M or M’
  – Must refine M, M’
  – Repeat substitutability check
Containment

\[ U \equiv L(SM'_i) \]

Compatibility

\[ SM' \parallel A \models P \]
\[ \mathcal{M} \setminus M \models A \]

refine M, M' spurious?

SM' not substitutable, CE provided

Predicate Abstraction

C C' A

Provide SM' to developer

B M M' M \setminus M
Feedback to the Developer

- If SM’ is substitutable
  - LKS showing how SM’ differs from M’

- If SM’ is not substitutable
  - counterexample showing the erroneous behavior in M’
Related Work

• Learning Assumptions: Cobleigh. et. al.
  – Do not consider *state labeling* of abstract models
  – Do not incorporate a **CEGAR framework** for AG

• Compatibility of Component Upgrades: Ernst et. al.
  – Do not consider *temporal sequence* of actions in generating invariants

• Interface Automata: Henzinger et. al.
  – Do not have **CEGAR, AG**
  – No procedure for *computing* interface automata
Experiments

• Prototype implementation in ComFoRT framework

• ABB IPC component assembly
  – modified WriteMessageToQueue component
  – checked for substitutability inside the assembly of four components (read, write, queue, critical section)
ComFoRT: Component Formal Reasoning Framework

- CCL
- Specs using State/Event
- Component Substitutability
- Compositional Reasoning
- Deadlock Detection
- LKS
- State/Event Temporal Logic

System Design → High-level Specification → Formal Model → Temporal Properties → Model Checker

- BUG FOUND
- DESIGN CORRECT