An Assume-Guarantee Method for Modular Verification of Evolving Component-Based Software

Pham Ngoc Hung, Nguyen Truong Thang, and Takuya Katayama

Japan Advanced Institute of Science and Technology – JAIST

{hungpn, thang, katayama}@jaist.ac.jp

Contents

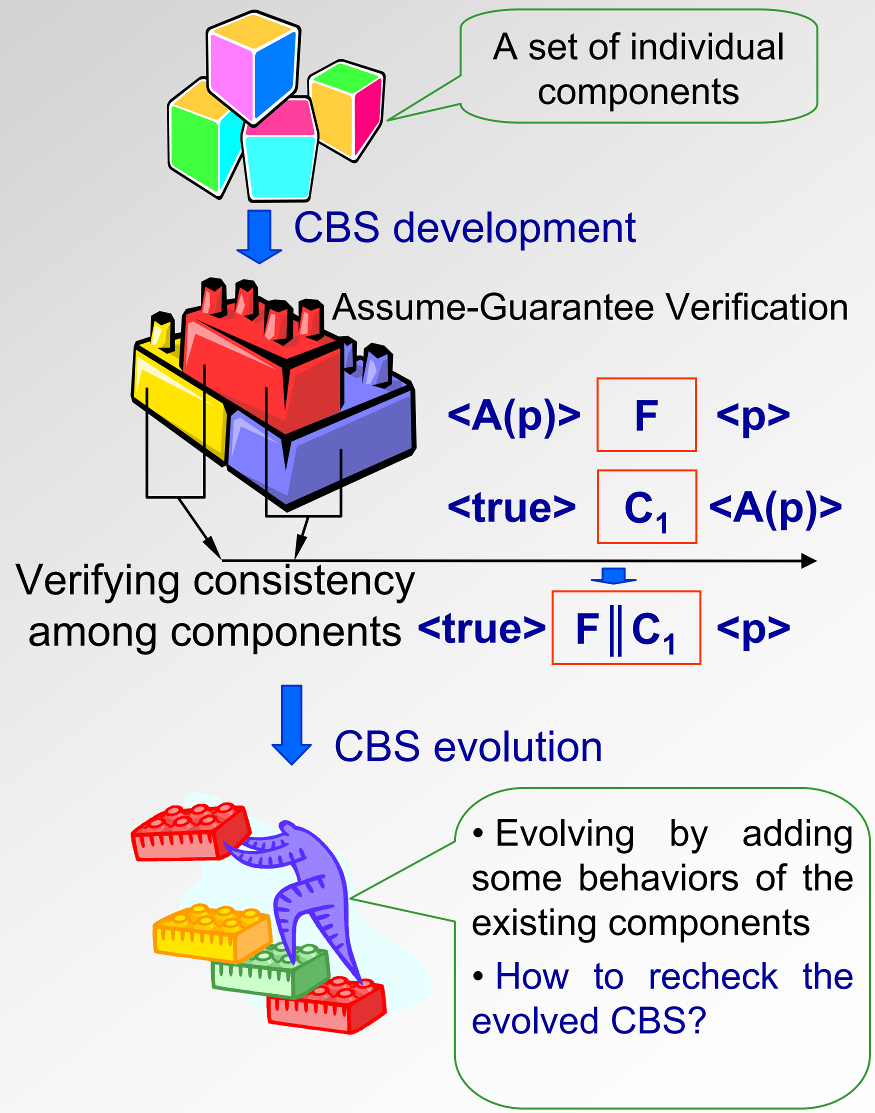
- **Introduction**
- Background
- A Framework for Modular Verification of Evolving CBS
- Assumption Regeneration Method
- Related Work & Conclusion

Component-Based Software (CBS)

- Structured from a set of well-defined components
  - Ideally, components are plug-and-play
  - Advantages: low development cost and time, flexible for changes, etc.
- One of key issues of CBS is "*component consistency*"
  - The currently well-known technologies as CORBA, COM/DCOM or .NET, JavaBeans and EJB (Sun), etc. only support "*component plugging*" -> plug-and-play mechanism often fails
  - **A potential solution: modular verification based on assume-guarantee reasoning**

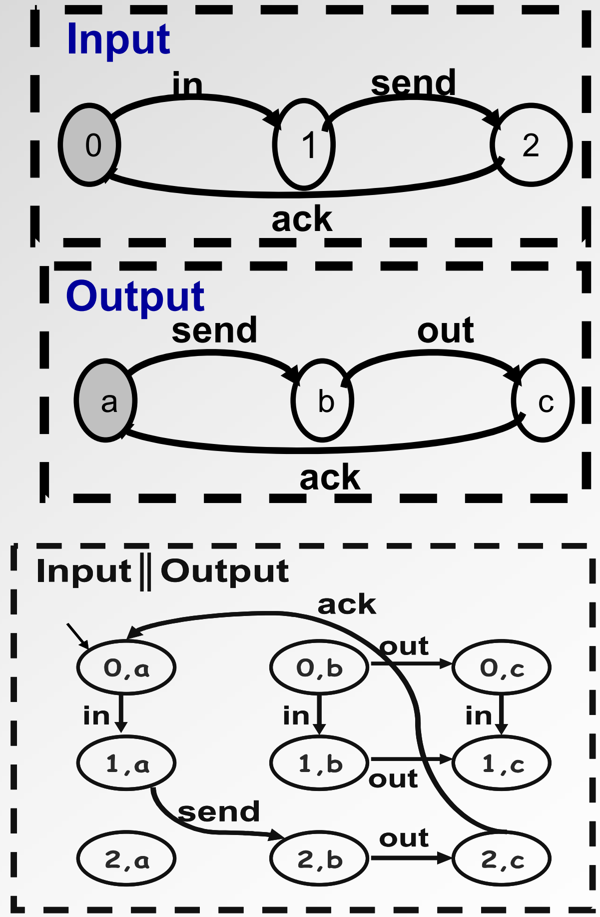
Evolving CBS

- CBS evolution seems to be an unavoidable task
  - Bug fixing, adding or removing some features, etc.
  - -> the whole evolved CBS must be rechecked
- How to recheck the evolved CBS by reusing the previous verification results?



Background (1/3)

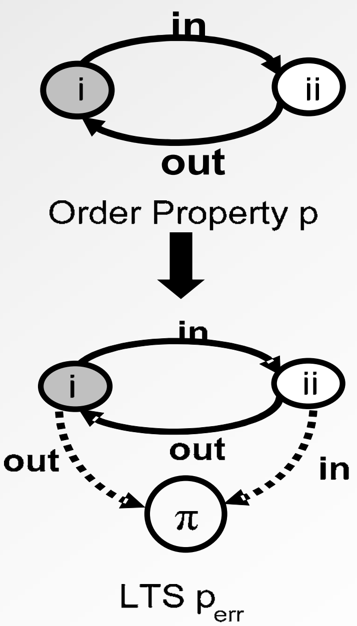
- Labeled Transition Systems (LTSs)
  - A LTS M = $\langle Q, \alpha M, \delta, q_0 \rangle$
- Parallel Composition Operator "‖"
  - Synchronizing the common actions
  - Interleaving the remaining actions
- Safety LTS, Safety Property, Satisfiability
  - A safety LTS: a deterministic LTS that contain no π state (π denotes the special error state)
  - A safety property is specified as a safety LTS p
  - A LTS M satisfies p ($M \models p$) iff $\forall \delta \in L(M)$: $(\delta \uparrow \alpha p) \in L(p)$



Background (2/3)

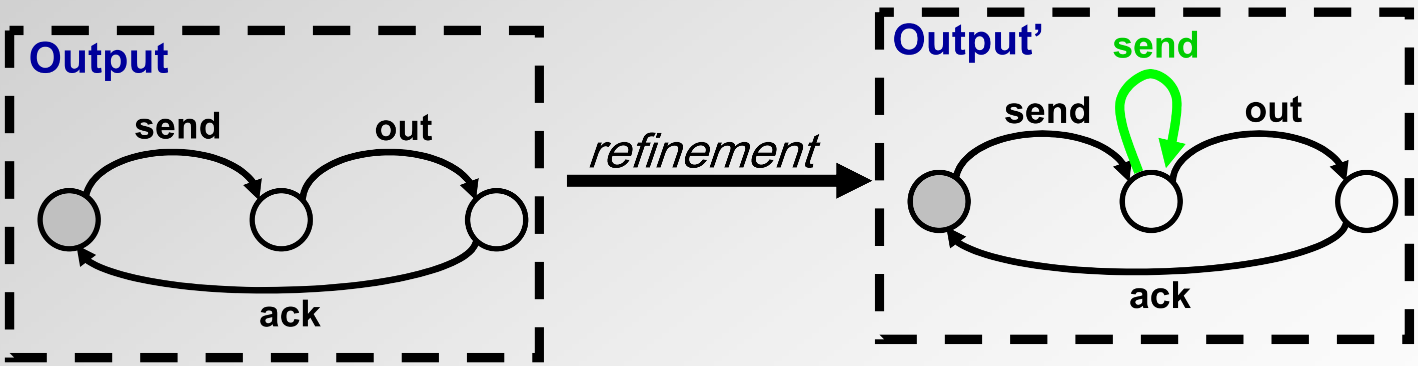
$$\frac{\begin{array}{ll}1. & \langle A(p)\rangle\ F\ \langle p\rangle \\ 2. & \langle \mathit{true}\rangle\ C_1\ \langle A(p)\rangle\end{array}}{\langle \mathit{true}\rangle\ F \mid\mid C_1\ \langle p\rangle}$$

- Assume-guarantee reasoning
  - *"Divide and conquer mechanism"* for decomposing a verification task into subtasks about the individual components of software
  - <A(p)> F <p>, <true> C_1 <A(p)> both hold -> $F \| C_1 \models p$
  - To check <A(p)> F <p>:
    1. Creating p_{err} from p: $\delta_{perr} = \delta_p \cup \{(q,a,\pi) \mid$ not exist $q' \in Q_p: (q,a,q') \in \delta_p\}$
    2. Computing $A(p) \| F \| p_{err}$
    3. If π is unreachable -> satisfied
  - Checking <true> C_1 <A(p)> by computing $C_1 \| A(p)_{err}$



Background (3/3)

- Component refinement
  - Adding some states and transitions into the old component
  - $C_1=<Q_1,\alpha C_1,\delta_1, q_0^1>$, $C_2=<Q_2,\alpha C_2,\delta_2, q_0^2>$: C_2 is the refinement of C_1 iff $Q_1 \subseteq Q_2$, $\delta_1 \subseteq \delta_2$, $q_0^1 = q_0^2$
    => $L(C_1) \subseteq L(C_2)$

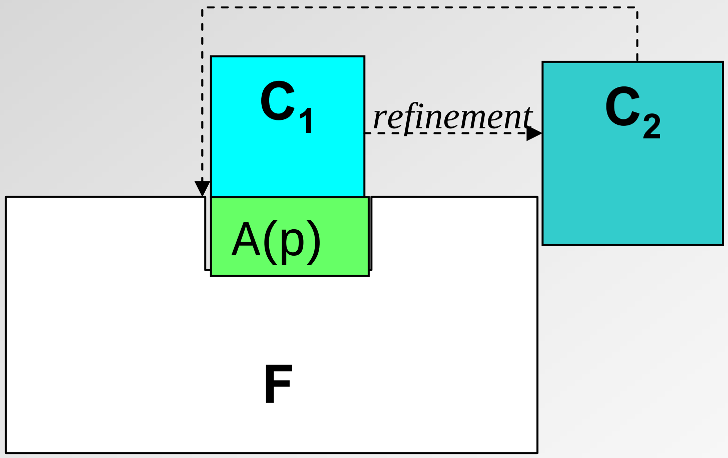


Contents

- Introduction
- Background
- **A Framework for Modular Verification of Evolving CBS**
- Assumption Regeneration Method
- Related Work
- Conclusion

Framework (1/2)

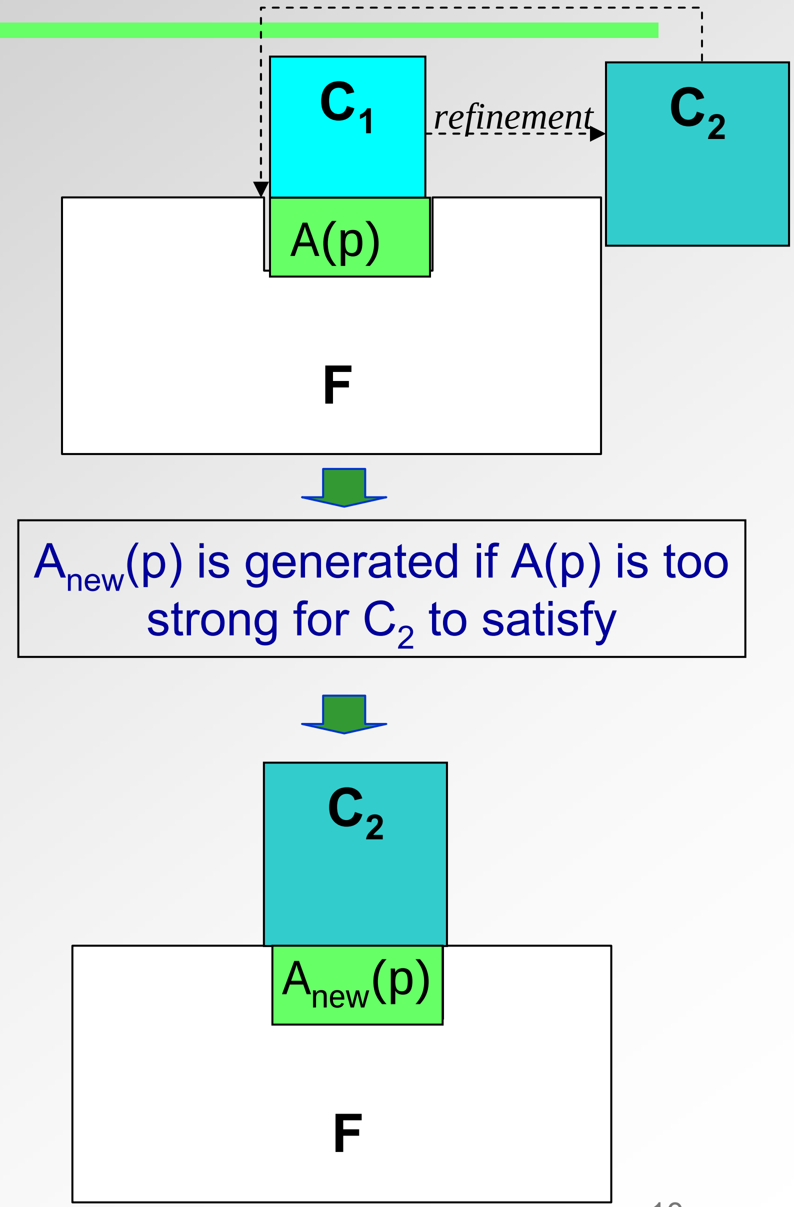
- Suppose that the system contains a framework F and an extension C_1 and $F \| C_1 \models p$
- Generating an assumption A(p)
  - Strong enough for F to satisfy p but weak enough to be discharged by C_1
  - *<A(p)>F<p> and <true>C_1<A(p)>* hold
  - When C_1 is *refined* into C_2
  - The goal: checking $F \| C_2 \models p$ by reusing the previous assumption A(p)



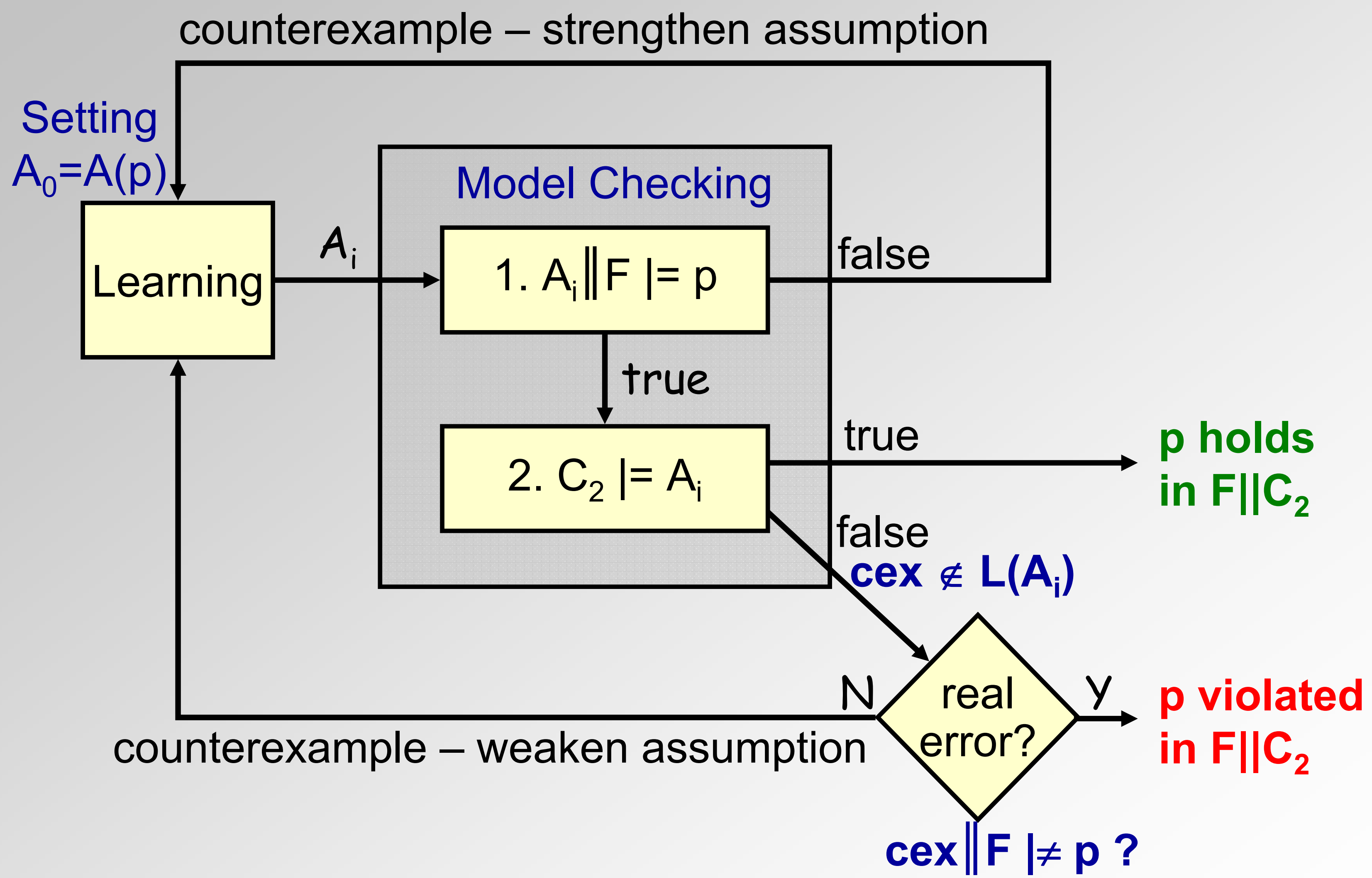
Framework (2/2)

- Solution
  - Only check ***$<true>C_2<A(p)>$***
  - If yes -> $F \| C_2 \models p$
  - Otherwise, $F \| C_2 \not\models p$ or A(p) is too strong for C_2 to satisfy
  - A new assumption $A_{new}(p)$ is re-generated by reusing A(p) if A(p) is too strong

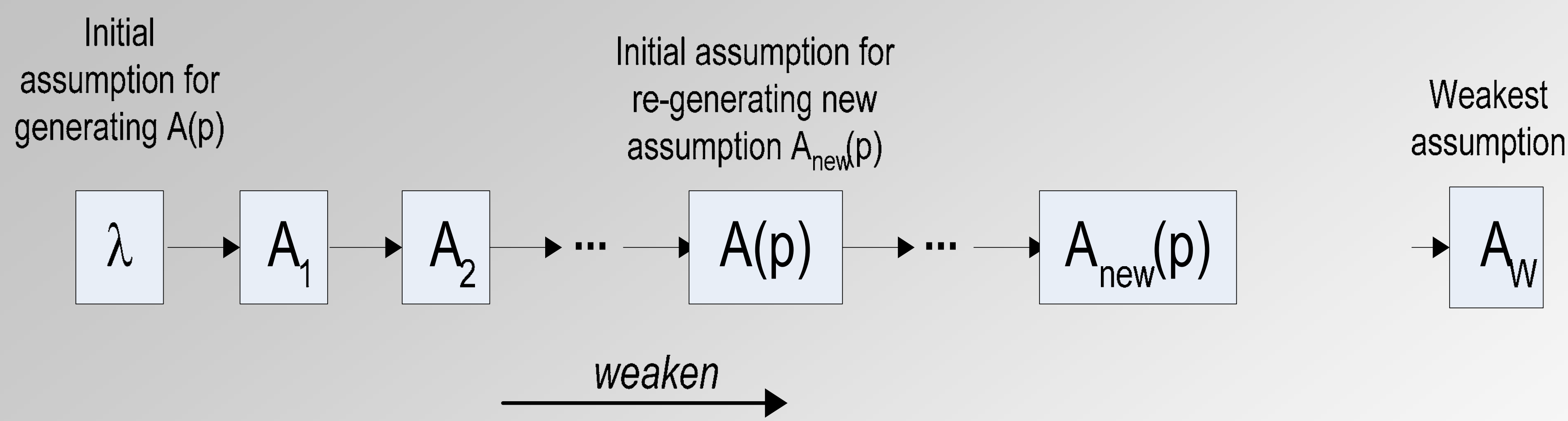
How to generate the new assumption $A_{new}(p)$?



Assumption regeneration process



Effectiveness



- To obtain the assumption $A_{new}(p)$, instead of starting from λ [Cobleigh'03], we start from the previous assumption A(p)
- This improvement reduces some steps of the assumption regeneration process

Correctness and termination

- **Theorem:** Given F, C_2 is a refinement of C_1, a property p and an assumption A(p): <A(p)>F<p>, <true>C_1<A(p)>. The process terminates and returns A_{new}(p) if $F \| C_2 \models p$ and false otherwise
  - Correctness
    - ✓ Guaranteed by the compositional rule
    - ✓ Always achieving A_{new}(p) by starting from A(p)
      - $C_2 \not\models A(p)$ and $C_2 \models A_{new}(p)$ -> A_{new}(p) is **weaker** than A(p)
  - Termination
    - ✓ At any iteration, it returns true or false and terminates or continues by providing a counterexample to L* Learning
    - ✓ $|A_0| \leq |A_1| \leq \ldots \leq |A_W|$
    - ✓ In the worst case: L* Learning produces A_W -> terminates!

Related Work

- Assume-guarantee verification [Cobleigh'03]
  - The basic case: two components C_1, C_2
  - Assumption generation by using L* algorithm
- Verification of evolving software [Sharygina'05]
  - Key idea: component substitutability analysis
    - ✓ Containment check: all local behavior of the old component contained in new one
    - ✓ Compatibility check: safety w.r.t other components in assembly
- OIMC [Thang&Katayama'04]
  - Focus on the interaction between two components **Base** and **Extension**
  - Deriving a set of preservation constraints at the interface states of Base

Conclusion

- A framework for evolving CBS verification in the context of component refinement
- An assumption regeneration method
  - Reuse the previous assumption
  - Reduce several steps of the process
- Future work
  - Evaluating the effectiveness formally
  - Applying the method for some larger case studies

Thanks for your listening!

Assume-guarantee verification [Cobleigh'03]

- The main ideas base on **Assume-Guarantee**
- The system has only two components; M_1, M_2
- The main goal: checking $\mathbf{M_1 \| M_2 \models p}$ *without composing M_1 with M_2?*
- Finding an assumption A satisfying the compositional rule by using L*
- If these components are changed -> assumption generation process re-runs on the whole system from beginning

Verification of evolving software [Sharygina'05]

- Key idea: component substitutability analysis
  - Obtain a finite behavioral model of all components by abstraction
  - Containment check: all local behavior of the old component contained in new one
    - ✓ Use under- and over- approximations
  - Compatibility check: safety w.r.t other components in assembly
    - ✓ Use dynamic assume-guarantee reasoning (dynamic L*)

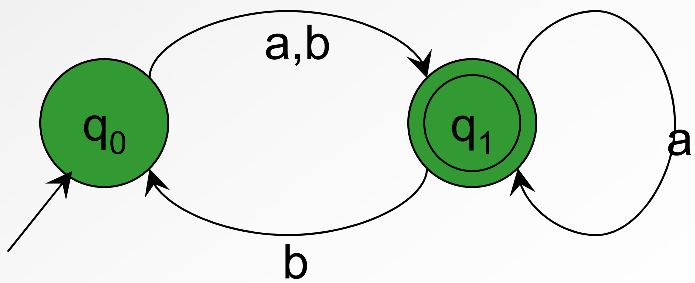
Verification of evolving software [Sharygina'05]

- Component refinement: adding and removing some behavior of component implementation
- Using abstraction to obtain a new model of the upgraded component
- Try to reuse the old assumption to verify the new system by improving L* -> dynamic L*
- Our opinion: adding is enough
- We want not only to reuse the previous assumptions but also to reuse the previous models

Learning algorithm - L*

- Proposed by D. Angluin, improved by Rivest
- learns an unknown regular language **U**
- produces a **D**eterministic **F**inite state **A**utomata (**DFA**) **C** such that **L(C) = U** (the minimal DFA C corresponding to U)
- **DFA M = (Q, q^0, αM, δ, F) :**
  - **Q, q^0, αM, δ** : as in deterministic LTS
  - **F** $\subseteq$ **Q** : accepting states
  - **L(M) = {σ | δ(q^0, σ) $\in$ F}**

aaa$\in$L(M), aaab$\notin$L(M)



A DFA example

The base idea of L*

Myhill-Nerode Theorem

For every regular set $U \subseteq \Sigma^*$ there exists a unique minimal deterministic automata whose states are isomorphic to the set of **equivalence classes** of the following relation:

$$w \approx w' \text{ iff } \forall u \in \Sigma^* : wu \in U \Leftrightarrow w'u \in U$$

Basic idea: learn the **equivalence classes**

- Two prefixes are not in the same class iff there is a **distinguishing suffix u**