Automated Component-Based Verification

Dimitra Giannakopoulou and Corina Păsăreanu
CMU / NASA Ames Research Center
component-based development

- component-based verification, for increased scalability, at design level
- early detection of integration problems
- use design level artifacts to improve/aid coding and testing

![Diagram showing component-based development process]

- cost of detecting/fixing defects increases
- integration issues handled early
## structure

<table>
<thead>
<tr>
<th>part 1 (Dimitra)</th>
<th>part 2 (Corina)</th>
</tr>
</thead>
<tbody>
<tr>
<td>assume-guarantee reasoning</td>
<td>multiple components</td>
</tr>
<tr>
<td>computing assumptions</td>
<td>alphabet refinement</td>
</tr>
<tr>
<td>learning assumptions</td>
<td>case studies</td>
</tr>
<tr>
<td>discussion</td>
<td>discussion</td>
</tr>
</tbody>
</table>

### lunch

<table>
<thead>
<tr>
<th>part 3 (Dimitra)</th>
<th>part 4 (Corina)</th>
</tr>
</thead>
<tbody>
<tr>
<td>component interfaces</td>
<td>reasoning about code</td>
</tr>
<tr>
<td>compositional JavaPathfinder</td>
<td>abstraction</td>
</tr>
<tr>
<td>examples</td>
<td>related work</td>
</tr>
<tr>
<td>discussion</td>
<td>conclusion</td>
</tr>
</tbody>
</table>
void add(Object o) {
    buffer[head] = o;
    head = (head+1)%size;
}

Object take() {
    ...= (tail+1)%size;
    return buffer[tail];
}

model checking

program / model

model checker

property

always(\phi \lor \psi)

YES (property holds)

NO + counterexample: (provides a violating execution)
model checking vs. testing

testing

model checking
compositional verification
<table>
<thead>
<tr>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prof. Howard Barringer (Univ. of Manchester)</td>
</tr>
<tr>
<td>Colin Blundell (Upenn, IBM Research)</td>
</tr>
<tr>
<td>Jamieson Cobleigh (UMass, MathWorks)</td>
</tr>
<tr>
<td>Michael Emmi (UCLA)</td>
</tr>
<tr>
<td>Mihaela Gheorgiu (Univ. of Toronto, JPL)</td>
</tr>
<tr>
<td>Chang-Seo Park (UC Berkeley)</td>
</tr>
<tr>
<td>Suzette Person (Univ. of Nebraska, NASA Langley)</td>
</tr>
<tr>
<td>Rishabh Singh (MIT)</td>
</tr>
</tbody>
</table>
compositional verification

does system made up of $M_1$ and $M_2$ satisfy property $P$?

- check $P$ on entire system: too many states!
- use system’s natural decomposition into components to break-up the verification task
- check components in isolation:

$M_1$ satisfies $P$?

$M_2$
“when we try to pick out anything by itself, we find it hitched to everything else in the universe”

John Muir
assume-guarantee reasoning

introduces assumptions / reasons about triples:

\[ <A> M <P> \]

is true if whenever M is part of a system that satisfies A, then the system must also guarantee P

simplest assume-guarantee rule (ASYM):

1. \[ <A> M_1 <P> \]
2. \[ <true> M_2 <A> \]

\[ <true> M_1 \parallel M_2 <P> \]

“discharge” the assumption
examples of assumptions

- will not invoke “close” on a file if “open” has not previously been invoked
- accesses to shared variable “X” must be protected by lock “L”
- (rover executive) whenever thread “A” reads variable “V”, no other thread can read “V” before thread “A” clears it first
- (spacecraft flight phases) a docking maneuver can only be invoked if the launch abort system has previously been jettisoned from the spacecraft
how do we come up with the assumption?
formalisms

- components modeled as **finite state machines** (FSM)
  - FSMs assembled with parallel composition operator “||”
    - synchronizes shared actions, interleaves remaining actions
- a safety property P is a **FSM**
  - P describes all legal behaviors in terms of its alphabet
  - $P_{err}$ — complement of P
    - determinize & complete P with an “error” state;
    - bad behaviors lead to error
  - component M satisfies P iff error state unreachable in $(M || P_{err})$
- **assume-guarantee** reasoning
  - assumptions and guarantees are FSMs
  - $\langle A \rangle M \langle P \rangle$ holds iff error state unreachable in $(A || M || P_{err})$
require in and out to alternate (property Order)
parallel composition

Input

|   in   | send | ack   |

Output

| send   | out   | ack   |
crex. 1: \((I_0, O_0)\) out \((I_0, O_{error})\)

crex. 2: \((I_0, O_0)\) in \((I_1, O_1)\) send \((I_2, O_1)\) out \((I_2, O_0)\) out \((I_2, O_{error})\)
assume-guarantee reasoning

\[ \text{crex 1: } (I_0, A_0, O_0) \text{ out} \quad \text{X} \]
\[ \text{crex 2: } (I_0, A_0, O_0) \text{ in } (I_1, A_0, O_1) \text{ send } (I_2, A_1, O_1) \text{ out } (I_2, A_0, O_0) \text{ out} \quad \text{X} \]
given component M, property P, and the interface of M with its environment, generate the weakest environment assumption \texttt{WA} such that: \[\langle \texttt{WA} \rangle \ M \langle P \rangle \text{ holds}\]

weakest means that for all environments E:

\[\langle \texttt{true} \rangle \ M \parallel E \langle P \rangle \text{ IFF } \langle \texttt{true} \rangle \ E \langle \texttt{WA} \rangle\]
weakest assumption in AG reasoning

1. \langle A \rangle M_1 \langle P \rangle
2. \langle true \rangle M_2 \langle A \rangle
\hline
\langle true \rangle M_1 \parallel M_2 \langle P \rangle

weakest assumption makes rule complete

for all E, \langle true \rangle M \parallel E \langle P \rangle IFF \langle true \rangle E \langle WA \rangle

\langle WA \rangle M_1 \langle P \rangle holds (WA could be false)
\langle true \rangle M_2 \langle WA \rangle holds implies \langle true \rangle M_1 \parallel M_2 \langle P \rangle holds
\langle true \rangle M_2 \langle WA \rangle not holds implies \langle true \rangle M_1 \parallel M_2 \langle P \rangle not holds
STEP 1: composition, hiding, minimization

property true!
(all environments)

STEP 2: backward propagation of error along $\tau$ transitions

property false!
(all environments)

STEP 3: property extraction (subset construction & completion)

assumption
step 1: composition & hiding

\[ \text{Input} \parallel \text{Order}_{\text{err}} \{ \text{in} \} \]
step 2: error propagation
step 3: subset construction
step 3: subset construction
step 3: property construction
weakest assumption in AG reasoning

1. \langle A \rangle M_1 \langle P \rangle
2. \langle true \rangle M_2 \langle A \rangle

\langle true \rangle M_1 || M_2 \langle P \rangle

\langle WA \rangle M_1 \langle P \rangle holds (WA could be false)
\langle true \rangle M_2 \langle WA \rangle holds implies \langle true \rangle M_1 || M_2 \langle P \rangle holds
\langle true \rangle M_2 \langle WA \rangle not holds implies \langle true \rangle M_1 || M_2 \langle P \rangle not holds

weakest assumption makes rule complete
L* learns unknown regular language $U$ (over alphabet $\Sigma$) and produces minimal DFA $A$ such that $L(A) = U$ 
($L^*$ originally proposed by Angluin)
(queries) should word $w$ be included in $L(A)$?

Yes / no: $w$ should (not) be in $L(A)$

(conjectures) here is an $A$ – is $L(A) = U$?

Yes!

no: $w$ should (not) be in $L(A)$
oracle for WA in assume-guarantee reasoning

\[
\begin{align*}
\langle WA \rangle M_1 \langle P \rangle & \text{ holds} \\
\langle true \rangle M_2 \langle WA \rangle & \text{ holds implies } \langle true \rangle M_1 || M_2 \langle P \rangle \text{ holds} \\
\langle true \rangle M_2 \langle WA \rangle & \text{ does not hold implies } \langle true \rangle M_1 || M_2 \langle P \rangle \text{ does not hold}
\end{align*}
\]
characteristics

assumptions conjectured by $L^*$ are not comparable semantically

- terminates with *minimal* automaton $A$ for $U$
- generates DFA candidates $A_i$: $|A_1| < |A_2| < \ldots < |A|$
- produces at most $n$ candidates, where $n = |A|$
- # queries: $O(kn^2 + n \log m)$,
  - $m$ is size of largest counterexample, $k$ is size of alphabet
- for assume-guarantee reasoning, may terminate early with a smaller assumption than the weakest
we check: \(\langle \text{true} \rangle \text{ Input} \parallel \text{ Output} \langle \text{Order} \rangle\)

\(M_1 = \text{Input}, M_2 = \text{Output}, P = \text{Order}\)

assumption alphabet: \{send, out, ack\}
**queries**

### Table $T$

<table>
<thead>
<tr>
<th>$S$</th>
<th>$S \cdot \Sigma$</th>
<th>$E$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda$</td>
<td>ack</td>
<td>true</td>
</tr>
<tr>
<td>out</td>
<td>out, ack</td>
<td>false</td>
</tr>
<tr>
<td>out</td>
<td>out, out</td>
<td>false</td>
</tr>
<tr>
<td>out, send</td>
<td>out, send</td>
<td>false</td>
</tr>
</tbody>
</table>

**$S$ = set of prefixes**

**$E$ = set of suffixes**
candidate construction

\[ E = \text{set of suffixes} \]
\[ S = \text{set of prefixes} \]

**Table T**

<table>
<thead>
<tr>
<th></th>
<th>( \lambda )</th>
<th>( \text{true} )</th>
<th>( \text{false} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S )</td>
<td>( \lambda )</td>
<td>( \text{true} )</td>
<td>( \text{false} )</td>
</tr>
<tr>
<td></td>
<td>( \text{out} )</td>
<td>( \text{false} )</td>
<td></td>
</tr>
</tbody>
</table>

\[ S \cdot \Sigma \]

<table>
<thead>
<tr>
<th></th>
<th>( \text{ack} )</th>
<th>( \text{true} )</th>
<th>( \text{false} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{out} )</td>
<td>( \text{false} )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{send} )</td>
<td>( \text{false} )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{out}, \text{ack} )</td>
<td>( \text{true} )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{out}, \text{out} )</td>
<td>( \text{false} )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{out}, \text{send} )</td>
<td>( \text{false} )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Order\text{err}**

**Output**

- **Input**
  - in
  - send
  - ack

- **Order**
  - in
  - out

- **Output**
  - send
  - out

2 states – error state omitted

Assumption \( A_1 \)

Counterexamples add to \( S \)
conjectures

Oracle 1: \( \langle A_1 \rangle \) Input \( \langle \text{Order} \rangle \)

Counterexample: \( c = \langle \text{in}, \text{send}, \text{ack}, \text{in} \rangle \)

Return to \( L^* \): \( c \uparrow \Sigma = \langle \text{send}, \text{ack} \rangle \)

Oracle 2: \( \langle \text{true} \rangle \) Output \( \langle A_2 \rangle \) True

property \( \text{Order} \) holds on Input \( \| \) Output
please ask LOTS of questions!
Automated Component-Based Verification

part II

Dimitra Giannakopoulou and Corina Păsăreanu
CMU / NASA Ames Research Center
<table>
<thead>
<tr>
<th>recap from part I</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶ Compositional Verification</td>
</tr>
<tr>
<td>▶ Assume-guarantee reasoning</td>
</tr>
<tr>
<td>▶ Weakest assumption</td>
</tr>
<tr>
<td>▶ Learning framework for reasoning about 2 components</td>
</tr>
</tbody>
</table>
Does system made up of $M_1$ and $M_2$ satisfy property $P$?

- Check $P$ on entire system: too many states!
- Use the natural decomposition of the system into its components to break-up the verification task
- Check components in isolation:
  - Does $M_1$ satisfy $P$?
    - Typically a component is designed to satisfy its requirements in specific contexts / environments
  - Assume-guarantee reasoning:
    - Introduces assumption $A$ representing $M_1$’s “context”
assume-guarantee reasoning

- **Reason about triples:**
  \[\langle A \rangle \text{ M} \langle P \rangle\]
  The formula is *true* if whenever M is part of a system that satisfies A, then the system must also guarantee P

- **Simplest assume-guarantee rule – ASYM**

  1. \[\langle A \rangle \text{ M}_1 \langle P \rangle\]
  2. \[\langle \text{true} \rangle \text{ M}_2 \langle A \rangle\]

  \[\langle \text{true} \rangle \text{ M}_1 || \text{ M}_2 \langle P \rangle\]

  "discharge" the assumption

How do we come up with the assumption A?
(usually a difficult manual process)

**Solution:** synthesize A automatically
Given component M, property P, and the interface of M with its environment, generate the weakest environment assumption \( \text{WA} \) such that: \( \langle \text{WA} \rangle M \langle P \rangle \) holds

Weakest means that for all environments E:

\[
\langle \text{true} \rangle M \parallel E \langle P \rangle \iff \langle \text{true} \rangle E \langle \text{WA} \rangle
\]
STEP 1: composition, hiding, minimization

STEP 2: backward propagation of error along $\tau$ transitions

STEP 3: property extraction (subset construction & completion)

property true! (all environments)

property false! (all environments)

assumption
learning for assume-guarantee reasoning

- Use an off-the-shelf learning algorithm to build appropriate assumption for rule ASYM

\[
\begin{align*}
1. & \quad \langle A \rangle \quad M_1 \quad \langle P \rangle \\
2. & \quad \langle true \rangle \quad M_2 \quad \langle A \rangle \\
\hline
\langle true \rangle & \quad M_1 \parallel M_2 \quad \langle P \rangle
\end{align*}
\]

- Process is iterative
- Assumptions are generated by querying the system, and are gradually refined
- Queries are answered by model checking
- Refinement is based on counterexamples obtained by model checking
- Termination is guaranteed
learning assumptions

- Use $L^*$ to generate candidate assumptions
- $\alpha A = (\alpha M_1 \cup \alpha P) \cap \alpha M_2$

$\langle \text{query: string } s \rangle \rightarrow \langle s \rangle M_1 \langle P \rangle$  

$\langle \text{true} \rangle M_1 \langle A \rangle$  

$\langle \text{false} \rangle M_2 \langle A \rangle$  

$\langle \text{false} \rangle M_2 \langle A \rangle$  

$\langle \text{cex } c \rangle M_1 \langle P \rangle$  

$\langle \text{true} \rangle M_1 \langle A \rangle$  

$\langle \text{true} \rangle \rightarrow M_1 \langle P \rangle$  

$\langle \text{true} \rangle \rightarrow M_2 \langle A \rangle$  

$\langle \text{true} \rangle \rightarrow M_1 \langle P \rangle$  

Model Checking

1. $\langle A \rangle M_1 \langle P \rangle$
2. $\langle \text{true} \rangle M_2 \langle A \rangle$

$\langle \text{true} \rangle M_1 \parallel M_2 \langle P \rangle$

- Guaranteed to terminate
- Reaches weakest assumption or terminates earlier
part II

- compositional verification
- assume-guarantee reasoning
- weakest assumption
- learning framework for reasoning about 2 components

extensions:
- reasoning about $n > 2$ components
- symmetric and circular assume-guarantee rules
- alphabet refinement
To check if $M_1 || M_2 || \ldots || M_n$ satisfies $P$
- decompose it into $M_1$ and $M'_2 = M_2 || \ldots || M_n$
- apply learning framework recursively for 2$^{nd}$ premise of rule
- $A$ plays the role of the property

\[
\begin{align*}
1. & \langle A \rangle M_1 \langle P \rangle \\
2. & \langle true \rangle M_2 || \ldots || M_n \langle A \rangle \\
\hline
& \langle true \rangle M_1 || M_2 \ldots || M_n \langle P \rangle
\end{align*}
\]

At each recursive invocation for $M_j$ and $M'_j = M_{j+1} || \ldots || M_n$
- use learning to compute $A_j$ such that
  - $\langle A_i \rangle M_j \langle A_{j-1} \rangle$ is true
  - $\langle true \rangle M_{j+1} || \ldots || M_n \langle A_i \rangle$ is true
- Model derived from Mars Exploration Rover (MER) Resource Arbiter
  - Local management of resource contention between resource consumers (e.g. science instruments, communication systems)
  - Consists of $k$ user threads and one server thread (arbiter)

- Checked mutual exclusion between resources
  - E.g. driving while capturing a camera image are mutually incompatible

- Compositional verification scaled to >5 users vs. monolithic verification ran out of memory [SPIN’06]
recursive invocation

- Compute $A_1 \ldots A_5$ s.t.
  \[
  \langle A_1 \rangle U_1 \langle P \rangle
  \]
  \[
  \langle \text{true} \rangle U_2 \parallel U_3 \parallel U_4 \parallel U_5 \parallel \text{ARB} \langle A_1 \rangle
  \]
  \[
  \langle A_2 \rangle U_2 \langle A_1 \rangle
  \]
  \[
  \langle \text{true} \rangle U_3 \parallel U_4 \parallel U_5 \parallel \text{ARB} \langle A_2 \rangle
  \]
  \[
  \langle A_3 \rangle U_3 \langle A_2 \rangle
  \]
  \[
  \langle \text{true} \rangle U_4 \parallel U_5 \parallel \text{ARB} \langle A_2 \rangle
  \]
  \[
  \langle A_4 \rangle U_4 \langle A_3 \rangle
  \]
  \[
  \langle \text{true} \rangle U_5 \parallel \text{ARB} \langle A_4 \rangle
  \]
  \[
  \langle A_5 \rangle U_5 \langle A_4 \rangle
  \]
  \[
  \langle \text{true} \rangle \text{ARB} \langle A_5 \rangle
  \]

- Result:
  \[
  \langle \text{true} \rangle U_1 \parallel \ldots \parallel U_5 \parallel \text{ARB} \langle P \rangle
  \]
symmetric rules: motivation

$M_1 = \text{Input}, M_2 = \text{Output}, P = \text{Order}$

$A_1:$
ack \hspace{1cm} send

$A_2:$
send

$A_3:$
\text{ack}, \text{out}, \text{send}

$A_4:$
send

$M_1 = \text{Output}, M_2 = \text{Input}, P = \text{Order}$

$A_1:$
\text{in} \hspace{1cm} \text{ack}

$A_2:$
in

$A_3:$
\text{send}

$A_4:$
\text{ack, out, send}
**symmetric rules**

- Assumptions for both components at the same time
  - Early termination; smaller assumptions
- Example symmetric rule – \( \text{SYM} \)

\[
\begin{align*}
1. & \langle A_1 \rangle M_1 \langle P \rangle \\
2. & \langle A_2 \rangle M_2 \langle P \rangle \\
3. & L(\text{co}A_1 \ | | \ \text{co}A_2) \subseteq L(P) \\
\implies & \langle true \rangle M_1 \ | | M_2 \langle P \rangle
\end{align*}
\]

- \( \text{co}A_i = \) complement of \( A_i \), for \( i=1,2 \)
- Requirements for alphabets:
  - \( \alpha P \subseteq \alpha M_1 \cup \alpha M_2; \quad \alpha A_i \subseteq (\alpha M_1 \cap \alpha M_2) \cup \alpha P, \text{ for } i =1,2 \)
- The rule is sound and complete
- Completeness needed to guarantee termination
- Straightforward extension to \( n \) components

Ensure that any common trace ruled out by both assumptions satisfies \( P \).
learning framework for rule SYM

\[ \text{L\textsuperscript{*}} \]

\[ \langle A_1 \rangle M_1 \langle P \rangle \]

\[ \langle A_2 \rangle M_2 \langle P \rangle \]

\[ L(\text{coA}_1 \parallel \text{coA}_2) \subseteq L(P) \]

\[ \text{counterex. analysis} \]

P holds in \( M_1 \parallel M_2 \)

P violated in \( M_1 \parallel M_2 \)
circular rule

- Rule **CIRC** – from [Grumberg&Long – Concur’91]

1. \( \langle A_1 \rangle M_1 \langle P \rangle \)
2. \( \langle A_2 \rangle M_2 \langle A_1 \rangle \)
3. \( \langle true \rangle M_1 \langle A_2 \rangle \)

\[ \langle true \rangle M_1 \parallel M_2 \langle P \rangle \]

- Similar to rule ASYM applied recursively to 3 components
  - First and last component coincide
  - Hence learning framework similar
- Straightforward extension to \( n \) components
Rule ASYM
- Assumption alphabet was fixed during learning
- \( \alpha A = (\alpha M_1 \cup \alpha P) \cap \alpha M_2 \)

[SPIN’06]: A subset alphabet
- May be sufficient to prove the desired property
- May lead to smaller assumption

How do we compute a good subset of the assumption alphabet?
Solution – iterative alphabet refinement
- Start with small alphabet
- Add actions as necessary
- Discovered by analysis of counterexamples obtained from model checking
1. Initialize $\Sigma$ to subset of alphabet $\alpha A = (\alpha M_1 \cup \alpha P) \cap \alpha M_2$

2. If learning with $\Sigma$ returns true, return true and go to 4. (END)

3. If learning returns false (with counterexample $c$), perform extended counterexample analysis on $c$.

   - If $c$ is real, return false and go to 4. (END)
   - If $c$ is spurious, add more actions from $\alpha A$ to $\Sigma$ and go to 2.

4. END
extended counterexample analysis

\[ \alpha A = (\alpha M_1 \cup \alpha P) \cap \alpha M_2 \]

\[ \Sigma \subseteq \alpha A \] is the current alphabet

Refiner: compare \( c \uparrow \alpha A \) and \( t \uparrow \alpha A \)

Add actions to \( \Sigma \) and restart learning
alphabet refinement

\[ \Sigma = \{ \text{out} \} \quad \alpha A = \{ \text{send, out, ack} \} \]

\[ \langle \text{true} \rangle \text{ Output } \langle A_i \rangle \rightarrow \text{false with } c = \langle \text{send, out} \rangle \]

\[ c^{\uparrow \Sigma} = \langle \text{out} \rangle \]

\[ c^{\uparrow \alpha A} = \langle \text{send, out} \rangle \]

\[ \langle c^{\uparrow \Sigma} \rangle \text{ Input } \langle P \rangle \rightarrow \text{false with counterex. } t = \langle \text{out} \rangle \]

\[ \langle c^{\uparrow \alpha A} \rangle \text{ Input } \langle P \rangle \rightarrow \text{true} \]

compare \langle \text{out} \rangle \text{ with } \langle \text{send, out} \rangle \rightarrow \text{add “send” to } \Sigma
characteristics

- Initialization of $\Sigma$
  - Empty set or property alphabet $\alpha P \cap \alpha A$
- Refiner
  - Compares $t \uparrow \alpha A$ and $c \uparrow \alpha A$
  - Heuristics:
    - AllDiff adds all actions in the symmetric difference of the trace alphabets
    - Forward scans traces in parallel forward adding first action that differs
    - Backward symmetric to previous
- Termination
  - Refinement produces at least one new action and the interface is finite
- Generalization to $n$ components
  - Through recursive invocation

- See also learning with optimal alphabet refinement
  - Developed independently by Chaki & Strichman 07
Implementation in the LTSA tool
- Learning using rules ASYM, SYM and CIRC
- Supports reasoning about two and $n$ components
- Alphabet refinement for all the rules

Experiments
- Compare effectiveness of different rules
- Measure effect of alphabet refinement
- Measure scalability as compared to non-compositional verification

Extensions for
- SPIN
- JavaPathFinder
case studies

**Model of Ames K9 Rover Executive**
- Executes flexible plans for autonomy
- Consists of main *Executive* thread and *ExecCondChecker* thread for monitoring state conditions
- Checked for specific shared variable: if the *Executive* reads its value, the *ExecCondChecker* should not read it before the *Executive* clears it

**Model of JPL MER Resource Arbiter**
- Local management of resource contention between resource consumers (e.g. science instruments, communication systems)
- Consists of $k$ user threads and one server thread (arbiter)
- Checked mutual exclusion between resources
results

- Rule ASYM more effective than rules SYM and CIRC
- Recursive version of ASYM the most effective
  - When reasoning about more than two components
- Alphabet refinement improves learning based assume guarantee verification significantly
- Backward refinement slightly better than other refinement heuristics
- Learning based assume guarantee reasoning
  - Can incur significant time penalties
  - Not always better than non-compositional (monolithic) verification
  - Sometimes, significantly better in terms of memory
## Analysis Data

| Case    | \(|A|\) | Mem (MB) | Time (s) | \(|A|\) | Mem (MB) | Time (s) | Mem (MB) | Time (s) |
|---------|--------|----------|----------|--------|----------|----------|----------|----------|
| MER 2   | 40     | 8.65     | 21.90    | 6      | 1.23     | 1.60     | 1.04     | 0.04     |
| MER 3   | 501    | 240.06   | --       | 8      | 3.54     | 4.76     | 4.05     | 0.111    |
| MER 4   | 273    | 101.59   | --       | 10     | 9.61     | 13.68    | 14.29    | 1.46     |
| MER 5   | 200    | 78.10    | --       | 12     | 19.03    | 35.23    | 14.24    | 27.73    |
| MER 6   | 162    | 84.95    | --       | 14     | 47.09    | 91.82    | --       | 600      |
| K9 Rover| 11     | 2.65     | 1.82     | 4      | 2.37     | 2.53     | 6.27     | 0.015    |

\(|A|\) = assumption size  
Mem = memory (MB)  
Time (seconds)  
-- = reached time (30min) or memory limit (1GB)
please ask LOTS of questions!
Automated Component-Based Verification

part III

Dimitra Giannakopoulou and Corina Păsăreanu
CMU / NASA Ames Research Center
example: autonomous rendezvous and docking

- input provided as UML state-charts, properties of type:
  - “you need at least two operational sensors to proceed to next mode”
  - “your state estimator will only return good values if it has good readings from at least 2 sensors”

- 3 bugs detected

- scaling achieved with compositional verification:
  - non-compositional verification runs out of memory after exploring > 13M states
  - compositional verification terminates successfully in secs. Analyzes one component at a time. The largest assumption has 10 states and the largest component has 5947 states (so largest state space explored is less that 60K states, as opposed to 13M)
<table>
<thead>
<tr>
<th>structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>part 1 (Dimitra)</td>
</tr>
<tr>
<td>assume-guarantee reasoning</td>
</tr>
<tr>
<td>computing assumptions</td>
</tr>
<tr>
<td>learning assumptions</td>
</tr>
<tr>
<td>discussion</td>
</tr>
</tbody>
</table>

**lunch**

| part 3 (Dimitra) | part 4 (Corina) |
| component interfaces | reasoning about code |
| compositional JavaPathfinder | abstraction |
| examples | related work |
| discussion | conclusion |
recap in reverse order...

- assume-guarantee reasoning
- learning framework for 2 components
- weakest assumption
assume-guarantee reasoning

reasons about triples:

\[ \langle A \rangle M \langle P \rangle \]

is true if whenever \( M \) is part of a system that satisfies \( A \), then the system must also guarantee \( P \)

simplest assume-guarantee rule (\( ASYM \)):

1. \( \langle A \rangle M_1 \langle P \rangle \)
2. \( \langle true \rangle M_2 \langle A \rangle \)

\[ \langle true \rangle M_1 \parallel M_2 \langle P \rangle \]
Learning assumptions for AG reasoning

1. \( \langle A \rangle M_1 \langle P \rangle \)
2. \( \langle true \rangle M_2 \langle A \rangle \)
   
   \( \langle true \rangle M_1 || M_2 \langle P \rangle \)

\( c \uparrow \alpha A \)

Conjecture: \( A_i \)

Query: string \( s \)

\( \langle s \rangle M_1 \langle P \rangle \)

\( \langle A_i \rangle M_1 \langle P \rangle \)

\( \langle true \rangle M_2 \langle A_i \rangle \)

\( \langle true \rangle M_1 || M_2 \langle A_i \rangle \)

\( \langle true \rangle M_1 || M_2 \langle P \rangle \)

\( \langle true \rangle M_1 || M_2 \langle A \rangle \)
the weakest assumption

- given component $M$, property $P$, and the interface $\Sigma$ of $M$ with its environment, generate the **weakest** environment assumption $WA$ such that: $\langle WA \rangle M \langle P \rangle$ holds

- weakest means that for all environments $E$:
  
  $\langle true \rangle M || E \langle P \rangle \text{ IFF } \langle true \rangle E \langle WA \rangle$
STEP 1: composition, hiding, minimization

property true! (all environments)

STEP 2: backward propagation of error along $\tau$ transitions

property false! (all environments)

STEP 3: property extraction (subset construction & completion)

assumption
part III

- assume-guarantee reasoning
- learning framework for 2 components
- weakest assumption WA

- component interfaces / learning WA
- compositional JavaPathFinder
- examples and discussion
component interfaces

- beyond syntactic interfaces ("open" file before "close")
- document implicit assumptions

weakest assumption (WA):
- **safe**: accept NO illegal sequence of calls
- **permissive**: accept ALL legal sequences of calls
safety check
permissiveness check
permissiveness: the problem

\[ M_0 \xrightarrow{a} M_1 \xrightarrow{b} M_2 \xrightarrow{b} M_{\text{error}} \]

\[ A_0 \xrightarrow{a} A_1 \]

\[ A_0 \xrightarrow{b} A_{\text{error}} \]

\[ M_0 \xrightarrow{a} M_1 \xrightarrow{b} M_2 \xrightarrow{b} M_{\text{error}} \]

\[ A_0 \xrightarrow{a} A_1 \xrightarrow{b} A_{\text{error}} \]

\[ \tau^c \]
(queries) should word \( w \) be included in \( L(A) \)? yes / no

(conjectures) here is an \( A \) – is \( L(A) = U \)? yes!

(is \( A \) safe and permissive?) no: word \( w \) should (not) be in \( L(A) \)
queries (simulate / model check)

conjecture – safe (model check)

conjecture – permissive

Alur et al, 2005, Henzinger et al, 2005
our approach (Giannakopoulou & Pasareanu, FASE 2009)

model check for \((M_i, A_{\text{error}})\)

reached \((M_1, A_{\text{error}})\) by “a b”

query “a b”

no (“a b” should not be in A)

backtrack and continue search…
invoke a model checker within a model checker?
permissiveness check

MC: model check for \((M_i, A_{\text{error}})\)

reached \((M_i, A_{\text{error}})\) by trace \(t\)

if \((\text{memoized}(t) == \text{no})\) // \(t\) is spurious

backtrack and continue search

else // \(\text{memoized}(t) == \text{yes}\) or \(t\) not in memoized

model checker produces \(t\)

if \((\text{query}(t) == \text{yes})\)

return \(t\) to \(L^*\) // not permissive

else restart at MC
example

module M

Input

\[\text{in} \rightarrow \text{send} \rightarrow \text{ack} \rightarrow \text{out}\]

Order_{err}

\[\text{in} \rightarrow \text{out} \rightarrow \text{out} \rightarrow \text{in}\]

Output

\[\text{send} \rightarrow \text{out} \rightarrow \text{ack}\]

\[\langle \text{ack, out} \rangle ?\]

assumption learned for AG reasoning

\[\text{ack, out, send}\]

weakest assumption
A complete module for permissiveness check

- queries performed on Input $\parallel$ Order$_{err}$
- safety checked on Input $\parallel$ Order$_{err}$ $\parallel$ A$_{err}$
- permissiveness performed on Complete_Input $\parallel$ Order$_{err}$ $\parallel$ A$_{err}$

Check reachability of states:
(sink, *, error) or (*, non error, error)

\[ \langle \text{ack, out} \rangle : \ (\text{sink, error, error}) \]
in summary…

resolve non-determinism
dynamically & selectively
remember, it’s a heuristic
assume-guarantee reasoning

query: string $s$

false+crex $c$

conjecture: $A_i$

$L^*$

query $c \uparrow \alpha A$

$\langle A_i \rangle M_1 \langle P \rangle$

true

$\langle true \rangle M_2 \langle A_i \rangle$

true

false+crex $c$

$P$ holds in $M_1 || M_2$

true

false

$P$ violated in $M_1 || M_2$

only need permissiveness with respect to $M_2$!
assume-guarantee reasoning

interface generation / discharge

JavaPathfinder
UML statecharts

jpf-cv
http://babelfish.arc.nasa.gov/trac/jpf
UML framework in JPF

JPF supports model checking of UML state-machines with an approach that consists of three steps:

- translate the UML model into a corresponding Java program, using JPF’s state chart (sc) extension and application model
- choose model properties to verify, and configure verification tools accordingly
- optionally provide a guidance script that represents the environment of the model (external event sequence)
package ICSEITutorial;

import gov.nasa.jpf.sc.State;

public class Input extends State {
    class S0 extends State {
        public void input() {
            setState(s1);
        }
    } S0 s0 = makeInitial(new S0());

    class S1 extends State {
        public void send() {
            setState(s2);
        }
    } S1 s1 = new S1();

    class S2 extends State {
        public void acknowledge() {
            setState(s0);
        }
    } S2 s2 = new S2();
}
AG reasoning in JPF

JavaPathfinder (CVState.AutomatonState) → SCSafetyListener (SCSafetyAutomaton)
assumptions

▸ choiceGeneratorAdvanced
  – if selected action leads assumption to error state then do
    “vm.getSystemState().setIgnored(true)” (backtrack)

▸ instructionExecuted
  – advance automaton & set CVState.AutomatonState

▸ stateBacktracked
  – get CVState.AutomatonState
properties

- instructionExecuted
  - advance automaton & set CVState.AutomatonState
  - if automaton reaches error state, then check() returns false

- stateBacktracked
  - get CVState.AutomatonState

JavaPathfinder
(CVState.AutomatonState)

SCSafetyListener
SCSafetyAutomaton
interface generation in JPF

- queries and assumption safety checks
  - same as assume-guarantee reasoning
- assumption permissiveness check
  - requires special listener
conformance listener

- **executeInstruction**
  - if instruction to be executed is assertion violation, then perform “ti.skipInstruction()” (do not process exception) and “vm.getSystemState().setIgnored(true)” (backtrack)

- **instructionExecuted**
  - advance automaton & set CVState.AutomatonState
  - if automaton reaches error state, check memoized table (why?)

    - if counterexample stored and spurious, backtrack
    - else check() returns false

- **stateBacktracked**
  - get CVState.AutomatonState
boolean done = false;
while (!done){
   counterexample = null;

   ...}

SCConformanceListener assumption = new SCConformanceListener(
   new SCSafetyAutomaton(false, assume, alphabet_, "Assumption",
   CompleteModule , memoized_));
JPF jpf = createJPFInstance(assumption, property, CompleteModule);
jpf.run();

Path jpfPath = assumption.getCountercexample();
if (jpfPath != null){
   // nonerror in M & error in Aerr - this is what we are looking for
   counterexample = assumption.convert(jpfPath);
   if( query(counterexample)){ // cex is in L(A)
      done = true; // a real counterexample for L*
   } // otherwise you need to continue with your loop
} else
   done = true; // interface is permissive
}
package ICSETutorial;
import gov.nasa.jpf.sc.State;
import gov.nasa.jpf.cv.CVState;

public class InputWithProperty extends CVState {
    class S0 extends State {
        public void input() {
            setState(s1);
        }

        public void output() {
            assert(false);
        }
    }

    S0 s0 = makeInitial(new S0());

    ...
}

S0 = ( send -> S2
     | acknowledge -> S1),
S1 = ( output -> S1
     | send -> S1
     | acknowledge -> S1),
S2 = ( output -> S3
     | send -> S1),
S3 = ( send -> S1
     | acknowledge -> S0).
tool: JavaPathfinder

UML statechart model of the Ascent and EarthOrbit flight phases of a spacecraft

properties:
- “An event IsamRendezvous, which represents a docking maneuver with another spacecraft, fails if the LAS (launch abort system) is still attached to the spacecraft”
- “Event tliBurn (trans-lunar interface burn takes spacecraft out of the earth orbit and gets it into transition to the moon) can only be invoked if EDS (Earth Departure Stage) rocket is available”
Assumption 1: lasJetisson \rightarrow lasJetisson \rightarrow IsamRendezvous

Assumption 2: IsamRendezvous \rightarrow tliBurn \rightarrow IsamRendezvous

generated interface assumptions encode Flight Rules in terms of events
conclusions

- learning assumptions is not a panacea
  - may perform worse than monolithic verification
  - performs well when alphabets & assumptions are small
- computed interfaces may not be permissive
  - in our studies interfaces were satisfactory
  - there is more to say about this in part IV
- limited to statecharts
  - but wish to extend; it’s open source, help us!!!
- got funding
  - so expect a lot of activity on jpf-cv over the next year
Automated Component-Based Verification

part IV

Dimitra Giannakopoulou and Corina Păsăreanu
CMU / NASA Ames Research Center
reasoning about code
introducing abstraction
to reason about very large or infinite state spaces
related approaches
reasoning about code

Does $M_1 \parallel M_2$ satisfy $P$? Model check; build assumption $A$

Does $C_1 \parallel C_2$ satisfy $P$? Model check; use assumption $A$

[ICSE’2004] – good results but may not scale

Solution: replace model checking with testing!
introducing abstraction...

- apply predicate abstraction [Graf & Saidi, CAV 1997]
- apply learning to abstracted components
- use counterexamples to automatically refine abstractions as needed, using CEGAR (Counter-example Guided Abstraction Refinement) [Clarke et al., CAV 2000]

- **interfaces:** novel combination of under- and over- approximations with L* avoids exponentially expensive determinization step and generates minimal and precise interfaces [CAV 2010]

- implemented in ARMC model checker (and previously Magic)
- successfully applied to several benchmarks (Java2SDK library classes, OpenSSL)
CEGAR for compositional verification

- **CEGAR: counterexample guided abstraction refinement** – Clarke et al. 00
  - incremental construction of abstractions
  - abstractions are conservative
  - abstract counterexamples obtained may be spurious (due to over-approximation)
  - spurious counterexamples are used for abstraction refinement

- **two level compositional abstraction refinement** – Chaki et al. 03
  - analyze $C_1 || C_2 || \ldots || C_n \models P$
  - build finite-state abstractions: $A_1, A_2, \ldots, A_n$
  - minimize: $M_1, M_2, \ldots, M_n$
  - analyze: $M_1 || M_2 || \ldots || M_n \models P$?
  - refine based on counterexamples

- **permissive interfaces** – Henzinger et al. 05
  - uses CEGAR to compute interfaces
assume-guarantee abstraction refinement (AGAR)

- Challenge: instead of learning $A$, build $A$ as an abstraction of $M_2$

  1. $\langle A \rangle M_1 \langle P \rangle$
  2. $\langle \text{true} \rangle M_2 \langle A \rangle$

  $\langle \text{true} \rangle M_1 \parallel M_2 \langle P \rangle$

- build $A$ as an abstraction of $M_2$; $\langle \text{true} \rangle M_2 \langle A \rangle$ holds by construction

- check Premise 1: $\langle A \rangle M_1 \langle P \rangle$

- obtained counterexamples are analyzed and used to refine $A$

- variant of CEGAR with differences:
  - use counterexample from $M_1$ to refine abstraction of $M_2$
  - $A$ keeps information only about the interface (abstracts away the internal info)

- implemented in LTSA; combined with alphabet refinement;

- compares favorably with learning approach

- [CAV’08]
<table>
<thead>
<tr>
<th>Case</th>
<th>No alpha. ref.</th>
<th>Learning</th>
<th>With alpha. ref.</th>
<th>Learning</th>
<th>Sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AGAR</td>
<td></td>
<td>AGAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mem.</td>
<td>Time</td>
<td>Mem.</td>
<td>Time</td>
<td>Mem.</td>
</tr>
<tr>
<td>Gas Station</td>
<td>3 16</td>
<td>4.11</td>
<td>3.33</td>
<td>177</td>
<td>42.83</td>
</tr>
<tr>
<td></td>
<td>4 19</td>
<td>37.43</td>
<td>23.12</td>
<td>195</td>
<td>100.17</td>
</tr>
<tr>
<td></td>
<td>5 22</td>
<td>359.53</td>
<td>278.63</td>
<td>45</td>
<td>206.61</td>
</tr>
<tr>
<td>Chiron,</td>
<td>2 10</td>
<td>1.30</td>
<td>0.92</td>
<td>9</td>
<td>1.30</td>
</tr>
<tr>
<td>Property 2</td>
<td>3 36</td>
<td>2.59</td>
<td>5.94</td>
<td>21</td>
<td>5.59</td>
</tr>
<tr>
<td></td>
<td>4 160</td>
<td>8.71</td>
<td>152.34</td>
<td>39</td>
<td>27.1</td>
</tr>
<tr>
<td></td>
<td>5 4</td>
<td>55.14</td>
<td>-</td>
<td>111</td>
<td>569.23</td>
</tr>
<tr>
<td>Chiron,</td>
<td>2 4</td>
<td>1.07</td>
<td>0.50</td>
<td>9</td>
<td>1.14</td>
</tr>
<tr>
<td>Property 3</td>
<td>3 8</td>
<td>1.84</td>
<td>1.60</td>
<td>25 njm</td>
<td>4.45</td>
</tr>
<tr>
<td></td>
<td>4 16</td>
<td>4.01</td>
<td>18.75</td>
<td>45</td>
<td>25.49</td>
</tr>
<tr>
<td></td>
<td>5 4</td>
<td>52.53</td>
<td>-</td>
<td>122</td>
<td>134.21</td>
</tr>
<tr>
<td>MER</td>
<td>2 34</td>
<td>1.42</td>
<td>11.38</td>
<td>40</td>
<td>6.75</td>
</tr>
<tr>
<td></td>
<td>3 67</td>
<td>8.10</td>
<td>247.73</td>
<td>335</td>
<td>133.34</td>
</tr>
<tr>
<td></td>
<td>4 58</td>
<td>341.49</td>
<td>-</td>
<td>38</td>
<td>377.21</td>
</tr>
<tr>
<td>Rover Exec.</td>
<td>2 10</td>
<td>4.07</td>
<td>1.80</td>
<td>11</td>
<td>2.70</td>
</tr>
</tbody>
</table>
Check composition of software C components

\[ C_1 || C_2 \models P \]
Theorem:
An interface A permissive w.r.t C's must abstraction safe w.r.t C's may abstraction and is safe and permissive for C.
interface generation for infinite-state components

- conceptually simple and elegant
- expensive learning restarts
- need of tighter integration of abstraction refinement with L*
- LearnReuse method
1. if checkSafe($\sigma$, $C_{must}$) $\neq$ null
2. return “no”
3. $cex = \text{checkSafe}(\sigma, C_{may})$
4. if $cex == null$
5. return “yes”
6. $\text{Preds} = \text{Preds} \cup \text{Refine}(cex)$
7. $\text{Query}(\sigma, C)$
Conjecture : Oracle 1

1. cex = checkSafe(A, Cmay)
2. if cex == null
3. invoke Oracle2
4. If Query(cex, C) == “no”
5. return cex to L*
6. else
7. goto 1
Conjecture: Oracle 2

1. `cex = checkPermissive(A, Cmust)`
2. `if cex == null`
3. `return A`
4. `if Query(cex, C) == “yes”`
5. `return cex to L*`
6. `else`
7. `goto l`
NASA case study

- NASA CEV 1.5 EOR-LOR mission
- 26 methods
- Only LearnReuse finished
- 74 predicates, 14 states
- 52 minutes
### Our Previous Work at a Glance

- Learning-based AG reasoning (TACAS 2003)
- Recursive application of simple rule for reasoning about $n > 2$ components (FMSD 2009)
- Symmetric and circular assume-guarantee rules (SAVCBS 2003, FMSD 2009)
- Learning with alphabet refinement (TACAS 2007)
- Learning assumptions for interface automata (FM 2008)
- Assume-guarantee abstraction refinement (CAV 2008)
- Interface generation in JPF (FASE 2009)
- Interface generation for large/infinite-state components (CAV 2010)
### Other Related Work

<table>
<thead>
<tr>
<th>(Interfaces)</th>
<th>(L* for AG reasoning)</th>
<th>(L* for NFAs &amp; liveness)</th>
<th>(L* for model extraction)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Margaria et al, 2007</td>
<td>Cobleigh et al, 2006</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
other related work

- minimal separating automaton for disjoint languages $L_1$ and $L_2$
  - accept all words in $L_1$
  - accept no words in $L_2$
  - have the least number of states

- assume-guarantee reasoning
  - minimal separating automaton for $L(M_2)$ and $L(M_1) \cap L(coP)$

- algorithms
  - Gupta at al. 07: query complexity exponential in the size of the minimal DFAs for the two input languages
  - Chen et al. 09: query complexity quadratic in the product of the sizes of the minimal DFAs for the two input languages. Use 3 valued DFAs

- compositional verification in symbolic setting (Alur et al. 05)
- learning omega-regular languages for liveness (Farzan et al. 08)
- learning non-deterministic automata (Bollig et al. 09)
thank you!
part IV

- parts I – III: reasoning about finite-state models

- reasoning about code

- introducing abstraction
to reason about very large or infinite state spaces

- related approaches
Does $M_1 \parallel M_2$ satisfy $P$? Model check; build assumption $A$

Does $C_1 \parallel C_2$ satisfy $P$? Model check; use assumption $A$

[ICSE’2004] – good results but may not scale

Solution: replace model checking with testing! [IET Software 2009]
abstraction

- Reduces large/infinite data domains into small/finite abstract domains; e.g. replace `int` with `{ZERO, POS, NEG}`
- Produces a finite state abstract model that operates on the abstract domain
- Abstraction maps
  - Concrete states to abstract states
  - Concrete transitions to abstract transitions
- Framework of abstract interpretation
May abstraction produces a finite over-approximation

Must abstraction produces a finite under-approximation
abstraction in compositional verification

- apply (predicate) abstraction [Graf & Saidi, CAV 1997]
- apply learning to abstracted components
- use counterexamples to automatically refine abstractions/assumptions as needed [Magic]

- use abstraction refinement as an alternative to learning for building assumptions [AGAR, CAV 2008]

- **interfaces**: novel combination of under- and over- approximations with L* avoids exponentially expensive determinization step and generates minimal and precise interfaces [CAV 2010]
- implemented in ARMC model checker
- successfully applied to several benchmarks (Java2SDK library classes, OpenSSL)
CEGAR: counterexample guided abstraction refinement [Clarke et al. 00]
- incremental construction of (may) abstractions
- abstract counterexamples obtained may be spurious (due to over-approximation)
- spurious counterexamples are used for abstraction refinement
CEGAR for compositional verification

- two level compositional abstraction refinement – Chaki et al. 03
  - analyze $C_1 \parallel C_2 \parallel \ldots \parallel C_n \models P$
  - build finite-state abstractions: $A_1, A_2, \ldots, A_n$
  - minimize: $M_1, M_2, \ldots, M_n$
  - analyze: $M_1 \parallel M_2 \parallel \ldots \parallel M_n \models P$
  - refine based on counterexamples

- permissive interfaces – Henzinger et al. 05
  - uses CEGAR to compute interfaces
learning-based compositional verification for C code

- Check composition of software C components $C_1 || C_2 |= P$
- $C_1, C_2$ are large/infinite state
Challenge: instead of learning $A$, build $A$ as an abstraction of $M_2$

\[
\begin{align*}
1. \quad & \langle A \rangle \quad M_1 \quad \langle P \rangle \\
2. \quad & \langle true \rangle \quad M_2 \quad \langle A \rangle \\
\hline
\langle true \rangle \quad M_1 \parallel M_2 \quad \langle P \rangle
\end{align*}
\]
assume-guarantee abstraction refinement (AGAR)

- Challenge: instead of learning $A$, build $A$ as an abstraction of $M_2$

$$
\begin{array}{c}
1. \langle A \rangle M_1 \langle P \rangle \\
2. \langle true \rangle M_2 \langle A \rangle \\
\hline
\langle true \rangle M_1 \parallel M_2 \langle P \rangle
\end{array}
$$

- build $A$ as an (may) abstraction of $M_2$; $\langle true \rangle M_2 \langle A \rangle$ holds by construction

- check Premise 1: $\langle A \rangle M_1 \langle P \rangle$

- obtained counterexamples are analyzed and used to refine $A$

- variant of CEGAR with differences:
  - use counterexample from $M_1$ to refine abstraction of $M_2$
  - $A$ keeps information only about the interface (abstracts away the internal info)

- implemented in LTSA; combined with alphabet refinement;

- compares favorably with learning approach

- [CAV’08]
### Table 1. Comparison of AGAR and learning for 2 components, with and without alphabet refinement.

<table>
<thead>
<tr>
<th>Case</th>
<th>k</th>
<th>No alpha. ref.</th>
<th>Learning</th>
<th>With alpha. ref.</th>
<th>Learning</th>
<th>Sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>AGAR</td>
<td></td>
<td>AGAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mem. Time</td>
<td>Mem. Time</td>
<td>Mem. Time</td>
<td>Mem. Time</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gas Station</td>
<td>3</td>
<td>16</td>
<td>4.11</td>
<td>3.33</td>
<td></td>
<td>117</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>19</td>
<td>37.43</td>
<td>23.12</td>
<td></td>
<td>195</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>22</td>
<td>359.53</td>
<td>278.63</td>
<td></td>
<td>45</td>
</tr>
<tr>
<td>Chiron,</td>
<td>2</td>
<td>10</td>
<td>1.30</td>
<td>0.92</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>Property 2</td>
<td>3</td>
<td>26</td>
<td>2.59</td>
<td>5.94</td>
<td></td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>36</td>
<td>8.71</td>
<td>152.34</td>
<td></td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>4</td>
<td>55.14</td>
<td></td>
<td></td>
<td>111</td>
</tr>
<tr>
<td>Chiron,</td>
<td>2</td>
<td>3</td>
<td>1.07</td>
<td>0.50</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>Property 3</td>
<td>3</td>
<td>8</td>
<td>1.84</td>
<td>1.60</td>
<td></td>
<td>25 n jmj</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>16</td>
<td>4.01</td>
<td>18.75</td>
<td></td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>4</td>
<td>52.53</td>
<td></td>
<td></td>
<td>122</td>
</tr>
<tr>
<td>MER</td>
<td>2</td>
<td>34</td>
<td>1.42</td>
<td>11.38</td>
<td></td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>67</td>
<td>8.10</td>
<td>247.73</td>
<td></td>
<td>335</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>58</td>
<td>341.49</td>
<td></td>
<td></td>
<td>38</td>
</tr>
<tr>
<td>Rover Exec.</td>
<td>2</td>
<td>10</td>
<td>4.07</td>
<td>1.80</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>9</td>
<td>2.62</td>
<td></td>
<td></td>
<td>2.07</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>6</td>
<td>2.46</td>
<td></td>
<td></td>
<td>3.30</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sizes</th>
<th>$M_1$</th>
<th>$P_{err}$</th>
<th>$M_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gas Station</td>
<td>1960</td>
<td>643</td>
<td></td>
</tr>
<tr>
<td>Chiron, Property 2</td>
<td>16464</td>
<td>1623</td>
<td></td>
</tr>
<tr>
<td>Chiron, Property 3</td>
<td>134456</td>
<td>3447</td>
<td></td>
</tr>
<tr>
<td>MER</td>
<td>804</td>
<td>5559</td>
<td></td>
</tr>
<tr>
<td>Rover Exec.</td>
<td>2030</td>
<td>129228</td>
<td></td>
</tr>
<tr>
<td>Rover Exec.</td>
<td>143</td>
<td>1270</td>
<td></td>
</tr>
<tr>
<td>Rover Exec.</td>
<td>6683</td>
<td>7138</td>
<td></td>
</tr>
<tr>
<td>Rover Exec.</td>
<td>307623</td>
<td>22886</td>
<td></td>
</tr>
</tbody>
</table>
Use predicate abstraction to build may and must abstractions of component C

\[ L_{safe}(C) = L_{err}(C) \]

Interface A is safe: \( L(A) \subseteq L_{safe}(C) \)

Interface A is permissive: \( L_{safe}(C) \subseteq L(A) \)

**Theorem:**

An interface A permissive w.r.t. C's must abstraction and safe w.r.t C's may abstraction is safe and permissive for C.
Correctness: If algorithm terminates then the returned interface $A$ is safe and permissive for $C$. 
interface generation for infinite-state components

- conceptually simple and elegant
- expensive learning restarts

- need of tighter integration of abstraction refinement with L*^
- LearnReuse method
Query(\(\sigma, C\))

1. if checkSafe(\(\sigma, C^{must}\)) \(!=\) null
2. return “no”
3. cex = checkSafe(\(\sigma, C^{may}\))
4. if cex == null
5. return “yes”
6. Preds = Preds U Refine(cex)
7. Query(\(\sigma, C\))

Gives answers consistent with C
Conjecture : Oracle 1

1. \( \text{cex} = \text{checkSafe}(A, \ C^{\text{may}}) \)
2. if cex == null
3. invoke Oracle2
4. If Query(cex, C) == “no”
5. return cex to L*
6. else
7. goto 1
Conjecture: Oracle 2

1. $\text{cex} = \text{checkPermissive}(A, C^{\text{must}})$
2. if $\text{cex} == \text{null}$
   3. return $A$
4. If $\text{Query(cex, C)} == \text{“yes”}$
   5. return $\text{cex to L^*}$
   6. else
   7. goto 1
NASA case study

- NASA CEV 1.5 EOR-LOR mission
- 26 methods
- Only LearnReuse finished
- 74 predicates, 14 states
- 52 minutes
our previous work at a glance

- learning-based AG reasoning (TACAS 2003)
- recursive application of simple rule for reasoning about $n > 2$ components (FMSD 2009)
- symmetric and circular assume-guarantee rules (SAVCBS 2003, FMSD 2009)
- learning with alphabet refinement (TACAS 2007)
- learning assumptions for interface automata (FM 2008)
- assume-guarantee abstraction refinement (CAV 2008)
- interface generation in JPF (FASE 2009)
- interface generation for large/infinite-state components (CAV 2010)
### Other Related Work

<table>
<thead>
<tr>
<th>(interfaces)</th>
<th>(L* for AG reasoning)</th>
<th>(L* for model extraction)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tkachuk et al, 2003</td>
<td>Cobleigh et al, 2006</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(L* for separating automata)</th>
<th>(L* for NFAs &amp; liveness)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Gupta et al, 2007</td>
<td>Bollig et al, 2009</td>
<td></td>
</tr>
<tr>
<td>Chen et al, 2009</td>
<td>Farzan et al, 2008</td>
<td></td>
</tr>
</tbody>
</table>
other related work

- minimal separating automaton for disjoint languages \( L_1 \) and \( L_2 \)
  - accept all words in \( L_1 \)
  - accept no words in \( L_2 \)
  - have the least number of states

- assume-guarantee reasoning
  - minimal separating automaton for \( L(M_2) \) and \( L(M_1) \cap L(coP) \)

- algorithms
  - Gupta at al. 07: query complexity exponential in the size of the minimal DFAs for the two input languages
  - Chen et al. 09: query complexity quadratic in the product of the sizes of the minimal DFAs for the two input languages. Use 3 valued DFAs

- compositional verification in symbolic setting (Alur et al. 05)
- learning omega-regular languages for liveness (Farzan et al. 08)
- learning non-deterministic automata (Bollig et al. 09)
Conclusion

- Compositional verification and assume-guarantee reasoning
- Techniques for automatic assumption generation and compositional verification
  - Finite state systems and safety properties
- Data abstraction to deal with very large/infinite state spaces
- Techniques are promising in practice

Future:
- Techniques for discovering good system decompositions
- Parallelization for increased scalability
- Beyond safety: liveness, timed properties, probabilistic reasoning
- Run-time analysis
- More?
thank you!