Abstraction and Compositional Verification

© Riccardo Sisto, Politecnico di Torino
Abstractions

• Is it possible to use model checking when K is infinite?
  ⇒ Lead the question $K \models f$ to an equivalent question that can be solved by model checking
  ⇒ I.E., find a finite model $K^*$ and a formula $f^*$ such that:
    $K^* \models f^* \iff K \models f$ (finitary abstraction)

• This approach can be useful even when K is not infinite
  ⇒ Abstractions let us operate on simpler models without losses
Abstractions

• Reference:
Abstractions

• An abstraction can be formalized as a function $\alpha$ such that $K^* = \alpha(K)$ and $f^* = \alpha(f)$

• An abstraction is **correctness preserving** (sound) when
  $$K^* \models f^* \Rightarrow K \models f$$

• An abstraction is **error preserving** (complete) when
  $$K^* \models f^* \Leftarrow K \models f$$

• An abstraction that is correctness preserving and error preserving is said **strongly preserving**
Abstractions

• Correctness preservation (soundness):
  – If we verify $f^*$ on $K^*$, we can conclude $f$ holds on $K$
  – False errors can be detected

• Error preservation (completeness):
  – If we find that $f^*$ is false on $K^*$, we can conclude that $f$ is also false on $K$
  – Errors can be missed
Abstractions

- Aim: get an abstract model that is finite, or smaller than the original one
  - The abstraction must map more states (more runs) onto the same abstract state (run)
  - This can typically be obtained by neglecting some aspects of the original model
Strongly preserving abstractions

• How to define a strongly preserving abstraction for property $f$?
  – It is enough that what is neglected in $K$ has no influence on $f$

• Example:
  – Consider only atomic propositions that occur in $f$
  – Map all the runs characterized by the same sequences of such propositions in a single abstract run
If $f$ includes only $xzero$ and $xpos$,

Run = $(xzero \ (\text{odd number of } xpos))$ *
Approximations

• An abstraction is **exact** if
  – For each run of $K$ there is a corresponding run of $K^*$
  – For each run of $K^*$ there is a corresponding *set of runs* of $K$

• The number of runs can be further reduced by applying abstractions that “approximate” $K$, i.e. such that
  – Some of the runs of $K$ are not represented
  – Some runs that do not occur in $K$ are represented
Over/Under Approximations

• K is “approximated” by a K* that abstractly represents:
  – a superset of the runs of K (over-approximation)
  – a subset of the runs of K (under-approximation)
Over/Under Approximations

• For any LTL formula $f$,
  – an over-approximation is correctness preserving (enables determining with certainty if $f$ is true)
  – an under-approximation is error preserving (enables determining with certainty if $f$ is violated)
OverApproximation Example

If f includes only xzero and xpos,

Run = (xzero (any number of xpos))^∗

xzero  xpos  xpos  xpos
   xone  xtwo
Abstractions in practice

- If concrete runs must be generated from abstract ones, all the advantages of abstractions are lost.

- Ideal capability: generate (runs of) $K^*$ directly from program or specification modeled by $K$, i.e.
  - if $K_S$ is the Kripke structure modeling the program or specification $S$
  - We need a transformation from $S$ to $S^*$ such that $\text{runs}(K_{S^*}) = \text{runs}(K^*)$

- In this way, however, it is difficult to get exact approximations.
Example: program abstraction with over-approximation

• Compute the set C of variables that influence $f$ (influence cone):
  – Initially, C is initialized with all the variables that influence the truth value of atomic propositions in $f$
  – Then, C is completed by adding all the variables used directly or indirectly for computing the values assigned to the variables in C (iterative algorithm).

• Eliminate from the program
  – all the variables not included in C.
  – all the instructions that do not modify the variables in C.

• Substitute all the decisions that depend on eliminated variables with non-deterministic choices
Example

\[
i = 0;\\j = 0;\\k = 0;\\while (i < MAX && j < MAX) \{\\\quad \text{if (vect1}[i] < vect2}[j])\\\quad \quad \text{vect3}[k++] = vect1[i++];\\\quad \text{else}\\\quad \quad \quad \text{vect3}[k++] = vect2[j++];\\\}\while (i < MAX)\\\quad \text{vect3}[k++] = vect1[i++];\\while (j < MAX)\\\quad \text{vect3}[k++] = vect2[j++];\\\text{putchar('\\n');}\\\text{for (i=0; i < MAX*2; i++)}\\\quad \text{printf("%d
", vect3[i]);}\\\]
\]

\[](k>=i && k>=j)
Counter Example Guided Abstraction and Refinement (CEGAR)

• Using correctness-preserving abstractions, we can proceed by stepwise refinements:

compute an initial abstraction $h$ and let $S^* = h(S)$, $f^* = h(f)$;

while $(MC(K_{S^*} \models f^* ) = \text{no} + \text{counterex})$
    inspect counterex;
    if (counterex is a valid counterexample of $K_S \models f$)
        then
            debug $S$
        else
            refine $h$
Abstract Interpretation

• Many program abstractions can be formalized by the abstract interpretation concept:
  – Define a data abstraction (mapping the values taken by variables onto abstract domains)
    • Example: sign: integer -> {0, pos, neg}
  – Generate the runs by interpreting the program on abstract domains

• Reference:
Example: Predicate Abstraction

• Is based on substituting variables with predicates about their value:
  – Example: want to focus only on the sign of x
    • substitute x with the predicate x<0

• The final result is a (good quality) over-approximation

• Reference:
  – S. Graf, and H. Saidi, “Construction of abstract state graphs with PVS”, *Proc. 9th CAV, LNCS 1254, 1997*
Example

• Verify \( \neg((x<0) \Rightarrow \diamond (x \geq 0)) \) on a program that references \( x \) only in the following statements:
  
  \[
  \begin{align*}
  x &= 0 \\
  x &= x + 1 \\
  x &= \text{MAX}
  \end{align*}
  \]

• \( x \) can be substituted by variable \( x_{\text{neg}} \) (true if \( x \) is negative)
• The abstraction mapping substitutes statements according to the table:

<table>
<thead>
<tr>
<th>xneg</th>
<th>F</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>x &lt; 0</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>x ≥ 0</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>x = 0</td>
<td>xneg=F</td>
<td>xneg=F</td>
</tr>
<tr>
<td>x++</td>
<td>xneg=F</td>
<td>xneg=T/F</td>
</tr>
<tr>
<td>x &gt; MAX</td>
<td>T/F</td>
<td>F</td>
</tr>
</tbody>
</table>

• The table can be built using the theory of abstract interpretation.
Example with Predicates on more variables

```c
int x=0, y=0, z=0;
while (1) {
    while (recv(s,b)) {
        add(b);
        x++;
    }
    if (y != x) {
        y = x;
        z++;
    }
    z--;    
}
```

Equality of x and y is tested in the if statement

We can substitute x and y by the predicate x==y
- **Substitution table:**

<table>
<thead>
<tr>
<th></th>
<th>F</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>xeqy</td>
<td>xeqy=F/T</td>
<td>xeqy=F</td>
</tr>
<tr>
<td>x++</td>
<td>xeqy=F</td>
<td></td>
</tr>
<tr>
<td>y=x</td>
<td>xeqy=T</td>
<td>xeqy=T</td>
</tr>
<tr>
<td>y!=x</td>
<td>T</td>
<td></td>
</tr>
</tbody>
</table>
int x=0, y=0, z=0;

while (1) {
    while (recv(s,b)) {
        add(b);
        x++;
    }
    if (y != x) {
        y = x;
        z++;
    }
    z--;
}

int xeqy=1, z=0;

while (1) {
    while (recv(s,b)) {
        add(b);
        if (xeqy)
            xeqy=F;
        else
            xeqy=nd(T,F);
    }
    if (!xeqy) {
        xeqy=T;
        z++;
    }
    z--;
}
Compositional Verification

• Complex Software has modular organization

• It is natural that verification follows the same modular approach:
  – Verify local properties in single modules
  – Verify that the conjunction of local properties implies a global property
  – Advantages: better scalability of verification
Compositional Verification

• Different approaches have been proposed:
  – compositional minimization
  – assume-guarantee paradigm

• Currently, automation of compositional verification techniques for concurrent/distributed systems is still poor

• Research is trying to make this technique more usable
The assume-guarantee paradigm

• This is a way to formally deal with modular systems:
  – For each module, specify,
    • What can be assumed about the module environment
    • What the module must guarantee (if assumption is true)
  – Define a logic for reasoning on modular specifications

• Reference:
Assume-guarantee

- The specification of a module takes the form:
  \[<\varphi> M <\psi>\]
  - “If the environment of M satisfies TL formula \( \varphi \), then M in this environment satisfies \( \psi \)”

- These expressions can be used to define a logic system.

- For instance, a valid inference rule is:
  \[
  \begin{align*}
  &<T> M <\varphi> \\
  &<\varphi> M' <\psi> \\
  &\hline
  &<T> M \parallel M' <\psi>
  \end{align*}
  \]
• Using just this inference rule,
  – verifying a global system property can be reduced to verifying a set of local module properties
  – provided there is a chain of causal relationships

• Example:
  – $<T> M1 || M2 || M3 <\psi>$ can be proved by proving $<T> M1 <\varphi_1>$, $< \varphi_1 > M2 < \varphi_2 >$, and $< \varphi_2 > M3 <\psi>$

• Difficulty:
  – Define modules and their local properties so that the global property can be inferred
  – verify an “open” model (without a model of the environment)
Compositional Verification and Abstractions

• The problem of finding the assumptions on the environment can be addressed together with the abstraction problem
  – An assumption on the environment is like an abstraction of the real environment
  – TL formula is equivalent to Kripke structure with runs that are all those that satisfy the formula
  – Verifying $<\varphi> M <\psi>$ means verifying that $M \models (\varphi \Rightarrow \psi)$ or that $(M \parallel K\varphi) \models \psi$

• Finding right assumption about environment means finding right abstraction for environment
The Refinement/Abstraction Relationship

• Defining the Refinement (abstraction) concept:
  – $P \preceq Q$ means that
    • $P$ is a refinement (implementation) of $Q$ (specification)
    • $Q$ is an abstraction of $P$
  – Formally, refinement can be defined as a relation between corresponding Kripke structures: $K'$ refines $K$ if
    • $AP(K') \supseteq AP(K)$ (all the AP in $K$ are defined in $K'$ too)
    • For each run $\pi' = s_0' s_1' ...$ of $K'$ there exists a run $\pi = s_0 s_1 ...$ of $K$ such that, $\forall i$ and $\forall P \in AP(K)$, $I'(s_i', P) = I(s_i, P)$
Example

\[
\begin{align*}
&x=0 \quad x=1 \quad x=2 \quad x=3 \quad \ldots \\
&xzero \quad xpos \quad xpos \quad xpos \\
&xone \quad xtwo \quad \quad \\
\end{align*}
\]
Stronger forms of refinement can be defined:

- K simulates K’ if K is an abstraction of K’ and exists a state-by-state correspondence between the two:
  - \( \text{AP}(K') \supseteq \text{AP}(K) \) (all the AP in K are defined in K’ too)
  - For each state \( s' \) of K’ there exists a corresponding state \( s \) of K such that,
    - \( \forall P \in \text{AP}(K), I'(s', P) = I(s, P) \)
    - For each transition \( \rho'(s', s_1') \) of K’ there is a corresponding transition \( \rho(s, s_1) \) of K such that \( s_1' \) and \( s_1 \) are corresponding states too.
  - This stronger refinement preserves any CTL* formula
Compositional Verification and Refinement

• Refinement guarantees some properties. For example:

\[(P \preceq Q) \land (Q \models \varphi) \Rightarrow P \models \varphi\]

Can verify a formula on Q and conclude that it holds on P

\[(P \preceq P') \land (Q \preceq Q') \Rightarrow (P||Q) \preceq (P'||Q')\]

\[(P \preceq P') \Rightarrow (P||P'') \preceq (P'||P'')\]

If I substitute an implementation with its abstract specification in a parallel composition I get an abstraction of the whole system
Compositional Verification of Refinement Relationship

• Assume we have a system
  – Made up of 2 modules
  – Each with its abstract version (high-level specification)

• Verify that $P || Q$ is a correct refinement of $P' || Q'$

![Diagram showing abstract specification and implementation with P', Q', P, Q]
Compositional Verification of Refinement Relationship

• If $P \preceq P'$ and $Q \preceq Q'$
  – because we have verified it
  – or because we have built $P'$ and $Q'$ as over-approximated abstractions starting from $P$ and $Q$

• Then we can conclude that $(P||Q) \preceq (P'||Q')$

• And any LTL/$\forall$CTL* property that can be verified on the abstract version holds on the concrete version too.
Problem

• One module can use other modules for doing its job
  – Refinement may not hold on modules in isolation:
Example

\[
x_1 = x_2 = 0; \quad \text{while (1) {}
    \text{ } x_1 += \text{nd(-1,0,1)};
    \text{send}(x_1);
    \text{rec}(x_2);
}\]

\[
x_1 = x_2 = 0; \quad \text{while (1) {}
    \text{ } x_2 += \text{nd(-1,0,1)};
    \text{rec}(x_1);
    \text{send}(x_2);
}\]

\[
x_1 = x_2 = p = 0; \quad \text{while (1) {}
    \text{ } x_1 += \text{nd}(x_2+p,x_2-p);
    \text{p} = x_2;
    \text{send}(x_1);
    \text{rec}(x_2);
}\]

\[
x_1 = x_2 = p = 0; \quad \text{while (1) {}
    \text{ } x_2 += \text{nd}(x_1+p,x_1-p);
    \text{p} = x_1;
    \text{rec}(x_1);
    \text{send}(x_2);
}\]
Example

\[x_1 = x_2 = 0;\]
while (1) {
  \[x_1 \leftarrow nd(-1,0,1);\]
  \[send(x_1);\]
  \[rec(x_2);\]
}

\[x_1 = x_2 = p = 0;\]
while (1) {
  \[x_1 \leftarrow nd(x_2+p,x_2-p);\]
  \[p \leftarrow x_2;\]
  \[send(x_1);\]
  \[rec(x_2);\]
}
Solution

In these cases we can use another inference rule:

\[(P || Q' \leq P') \land (P'||Q \leq Q') \Rightarrow (P||Q) \leq (P'||Q')\]
Example

```
x1=x2=0;
while (1) {
    x1 += nd(-1,0,1);
    send(x1);
    rec(x2);
}

\( \forall l \)

```

```
x1=x2=p=0;
while (1) {
    x1 += nd(x2+p,x2-p);
    p = x2;
    send(x1);
    rec(x2);
}
```

```
x1=x2=0;
while (1) {
    x2 += nd(-1,0,1);
    rec(x1);
    send(x2);
}
```
References
