Debugging Concurrent (Logic) Programs with Abstract Interpretation

Samir Genaim and Andy King

Funded by the Royal Society short visit grant 16385
Outline of this talk

The role of concurrency in search
  Generate-and-test search paradigm
  Test-and-generate search paradigm

User-interface issues in suspension analysis

Applying suspension analysis
  Bugs
  False positives
  Timing and Precision Results Summary

How the analysis works
Generate-and-test search paradigm

- generate – place all the queens on the chessboard in some configuration;
- test – check whether the configuration is safe, that is, whether any one of the queens can take one another;
- repeat generate and test, searching until either a solution is found or all configurations are exhausted.
Applying generate-and-test to n-queens

- if two queens occur in a row, then the configuration is unsafe;

\[
\begin{array}{cccc}
\times & & & \\
& \times & & \\
& & \times & \\
& & & \times
\end{array}
\]

\[
\pi = \begin{cases}
6 \mapsto 2 \\
5 \mapsto 4 \\
4 \mapsto 6 \\
3 \mapsto 1 \\
2 \mapsto 3 \\
1 \mapsto 5
\end{cases}
\]

\[
L = [5, 3, 1, 6, 4, 2]
\]
Applying generate-and-test to n-queens

- if two queens occur in a row, then the configuration is unsafe;
- if no queens occur in a row, then another row must have two queens, so the configuration is unsafe;

\[ \pi = \begin{cases} 6 \mapsto 2 \\ 5 \mapsto 4 \\ 4 \mapsto 6 \\ 3 \mapsto 1 \\ 2 \mapsto 3 \\ 1 \mapsto 5 \end{cases} \]

\[ L = [5, 3, 1, 6, 4, 2] \]
Applying generate-and-test to n-queens

- if two queens occur in a row, then the configuration is unsafe;
- if no queens occur in a row, then another row must have two queens, so the configuration is unsafe;
- exactly one queen occurs in each row;

\[ L = [5, 3, 1, 6, 4, 2] \]

\[ \pi = \left\{ \begin{array}{c}
6 \mapsto 2 \\
5 \mapsto 4 \\
4 \mapsto 6 \\
3 \mapsto 1 \\
2 \mapsto 3 \\
1 \mapsto 5 
\end{array} \right\} \]
Applying generate-and-test to n-queens

- if two queens occur in a row, then the configuration is unsafe;
- if no queens occur in a row, then another row must have two queens, so the configuration is unsafe;
- exactly one queen occurs in each row;
- each (safe) configuration is a mapping $[1, n] \rightarrow [1, n]$ from a row number to a column number;

$\pi = \begin{cases} 
6 &\mapsto 2 \\
5 &\mapsto 4 \\
4 &\mapsto 6 \\
3 &\mapsto 1 \\
2 &\mapsto 3 \\
1 &\mapsto 5 
\end{cases}$

$L = [5, 3, 1, 6, 4, 2]$
Applying generate-and-test to n-queens

- if two queens occur in a row, then the configuration is unsafe;
- if no queens occur in a row, then another row must have two queens, so the configuration is unsafe;
- exactly one queen occurs in each row;
- each (safe) configuration is a mapping $[1, n] \rightarrow [1, n]$ from a row number to a column number;
- each map is injective and surjective, hence a permutation.

$$\pi = \begin{pmatrix}
6 & \mapsto & 2 \\
5 & \mapsto & 4 \\
4 & \mapsto & 6 \\
3 & \mapsto & 1 \\
2 & \mapsto & 3 \\
1 & \mapsto & 5 \\
\end{pmatrix} \quad L = [5, 3, 1, 6, 4, 2]$$
main(Soln) :- perm([1, 2, 3, 4, 5, 6], Soln), safe(Soln).

perm([], []).
perm(Ls, [X|Xs]) :- select(X, Ls, Rs), perm(Rs, Xs).

select(X, [X|Xs], Xs).
select(X, [CN|CNs], [CN|Rs]) :- select(X, CNs, Rs).

safe([]).
safe([CN | CNs]) :- no_attack(CNs, CN, 1), safe(CNs).

no_attack([], _, _).
no_attack([CN|CNs], First_CN, Diff) :-
    diagonal(Diff, First_CN, CN), Next_Diff is Diff + 1,
    no_attack(CNs, First_CN, Next_Diff).

diagonal(Diff, First_CN, CN) :- Diff =\= abs(First_CN - CN).
Test-and-generate search paradigm

- **generate** – place one new queen on the chessboard to construct a configuration incrementally;
- **test** – check whether the new queen is safe as soon as it is placed on the board; discard partial configurations that are definitely unsafe.
- **repeat** incremental generation and testing, searching until either a solution is found or all configurations are exhausted.
main(Soln) :-
    length(Soln, 6),
    safe(Soln),
    perm([1,2,3,4,5,6], Soln).

:- block diagonal(?,-,?), diagonal(?,, -).
diagonal(Diff, First_CN, CN) :- Diff =\= abs(First_CN CN).

<table>
<thead>
<tr>
<th>n</th>
<th>G-and-T</th>
<th>T-and-G</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>30.313</td>
<td>0.107</td>
</tr>
<tr>
<td>11</td>
<td>&gt;60</td>
<td>0.063</td>
</tr>
<tr>
<td>12</td>
<td>&gt;60</td>
<td>0.380</td>
</tr>
<tr>
<td>13</td>
<td>&gt;60</td>
<td>0.176</td>
</tr>
<tr>
<td>14</td>
<td>&gt;60</td>
<td>3.300</td>
</tr>
<tr>
<td>15</td>
<td>&gt;60</td>
<td>2.659</td>
</tr>
<tr>
<td>16</td>
<td>&gt;60</td>
<td>21.808</td>
</tr>
</tbody>
</table>
Abstract interpretation schemes have been proposed by Bigot, Codish, Codognet, Winsborough, etc for checking that a program and goal cannot reduce to such a possibly problematic suspension state.

They simulate the operational semantics by tracing the execution of the program over a finite (though possibly large) collection of abstract states.

These schemes either return:

- “yes” – the program and goal *definitely* cannot reduce to a suspension state;
Related work

- Abstract interpretation schemes have been proposed by Bigot, Codish, Codognet, Winsborough, etc for checking that a program and goal cannot reduce to such a possibly problematic suspension state.

- They simulate the operational semantics by tracing the execution of the program over a finite (though possibly large) collection of abstract states.

- These schemes either return:
  - “yes” – the program and goal *definitely* cannot reduce to a suspension state;
  - “don’t know” – program and goal *may* reduce to a suspension.
User-interface issues

The programmer:

▶ should be able to activate analysis with minimal interaction;

Visit http://www.sci.univr.it/~genaim/www/susweb/bin/susweb.cgi to see how bottom-up analysis can address these user-interface problems.
User-interface issues

The programmer:

▶ should be able to activate analysis with minimal interaction;
▶ sometimes will need to carefully scrutinise the results;

Visit http://www.sci.univr.it/~genaim/www/susweb/bin/susweb.cgi to see how bottom-up analysis can address these user-interface problems.
User-interface issues

The programmer:

- should be able to activate analysis with minimal interaction;
- sometimes will need to carefully scrutinise the results;
- should not hesitate about applying the analysis even to the largest programs.

Visit http://www.sci.univr.it/~genaim/www/susweb/bin/susweb.cgi to see how bottom-up analysis can address these user-interface problems.
Bugs from Arizona and Kent

For bessel, the analysis inferred a call pattern of \textit{false} for the predicate \texttt{bessel}, the problem stemming from the clause:

\begin{verbatim}
bessel(0, X, Y1, Y2) :- Y2 = 0.0, j0(10, X, Y).
\end{verbatim}

For \texttt{queens\_control}, the analysis only inferred that a certain predicate, \texttt{perm}, will not suspend if its first argument is ground:

\begin{verbatim}
:- block perm_aux(-, ?, ?). perm_aux(? , -, ?).
perm_aux(D1, D2, D) :- D1 = D2, D = D1.
\end{verbatim}
A Bug from Argonne National Labs

For ssd, a call pattern of \textit{false} was inferred was traced to the following predicate:

\begin{verbatim}
next_play( Remaining, Board, History, D) :-
    Remaining = [] |
    length( Board, Len),
    First is (2 * Len) // 3,
    try_pent( [], Remaining, ..., History, D).
%next_play( [], _, History, D) :-
%  print_history( " SOLN ", History, D).
next_play( [], _, History, D).
\end{verbatim}
Bugs from Manchester Metropolitan University

The predicate `lhs_strip_DmTm` includes a debugging/error handling case that merely calls `pp (! flushes the buffer):

```
lhs_strip_DmTm([],_,_,_,_):-
    pp('ERROR {Dm,Tm} not found in PiSet')!.
```

This clause does ground its third, fourth and fifth arguments.

```
lhs_strip_DmTm([],_,C,D,E):-
    pp('ERROR {Dm,Tm} not found in PiSet')!,
    C := error, D := error, E := error.
```

It is arguably better practise to abort the computation by binding the output arguments to rogue values.
A false positive from Oregon/ICOT

For the program semigroup, non-suspension could only not be shown for the top-level predicate `main`:

```prolog
main(N) :-
    kernel(K),
    append([begin|K],[end|R],S),
    spawn(S,R,Out,[]),
    count(Out,N).
```

- The analysis infers that `spawn(S,R,Out,[])` will not suspend if both `S` and `R` are ground (correct but crude);
- Neither `S` nor `R` are ground at the time of the call (though `kernel(K)` binds `K` to a ground structure);
- `spawn` actually implements a form of pipelined filter where the input stream `S` is fed by the output stream `R`. 

Samir Genaim and Andy King
Debugging Concurrent Logic Programs with AI
## Timing and precision results table

<table>
<thead>
<tr>
<th>source</th>
<th>program</th>
<th>precision</th>
<th>time (msecs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>preds</td>
<td>blocks</td>
</tr>
<tr>
<td>Debray</td>
<td>combo</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>transp</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>deriv</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Foster</td>
<td>insert</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>btree</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Howe</td>
<td>entails</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Huntbach</td>
<td>colouring</td>
<td>42</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>spanning</td>
<td>76</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td>eight_puzzle</td>
<td>97</td>
<td>88</td>
</tr>
<tr>
<td>Johnson</td>
<td>PTMddd</td>
<td>319</td>
<td>316</td>
</tr>
<tr>
<td>Naish</td>
<td>queens</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>King</td>
<td>msort_control</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>queens_control</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>Tick</td>
<td>bestpath</td>
<td>20</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>pascal</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>semigroup</td>
<td>20</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>mastermind</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>nand</td>
<td>25</td>
<td>25</td>
</tr>
</tbody>
</table>

Samir Genaim and Andy King

Debugging Concurrent Logic Programs with AI
Monotonic and definite Boolean functions

- Let $Bool_X$ denote the set of propositional formulae over $X$.
- $Mon_X \subseteq Bool_X$ are those formulae which can be constructed only from $\lor$, $\land$ and $X$, ie, $X \land (Y \lor Z)$ where $X = \{W, X, Y, Z\}$.
- $Def_X \subseteq Bool_X$ are those formulae which are conjunctions of propositional Horn formulae, ie, $(W \leftarrow (X \land Y)) \land (Z \leftarrow true)$.

Now suppose $X = \{X, Y\}$. Let $model_X(X \land Y) = \{\{X, Y\}\}$, $model_X(X \lor Y) = \{\{X\}, \{Y\}, \{X, Y\}\}$ and $model_X(X \leftarrow Y) = \{\emptyset, \{X\}, \{X, Y\}\}$.

- $f \in Def_X$ iff $\forall M, M' \in model_X(f). M \cap M' \in model_X(f)$;
- $f \in Mon_X$ iff $\forall M \in model_X(f) \forall M \subseteq M' \subseteq X. M' \in model_X(f)$;
- Finally let $f_1, f_2 \in Bool_X$. $f_1 \models f_2$ iff $model_X(f_1) \subseteq model_X(f_2)$. 

Samir Genaim and Andy King

Debugging Concurrent Logic Programs with AI
Reordering compound goals without actually reordering

\[-\text{block } p(-, ?).\]
\[p(X, Z) :- Z = 1.\]

\[-\text{block } q(-, ?), q(?,-).\]
\[q(X, Y) :- \text{true}.\]

\[-\text{block } r(?,-).\]
\[r(Y, Z) :- Y = 2.\]

\[
d_i \in \text{Mon}_X \quad g_i \in \text{Bool}_X
\]
\[
d_1 = X \quad g_1 = Z
\]
\[
d_2 = X \land Y \quad g_2 = \text{true}
\]
\[
d_3 = Z \quad g_3 = Y
\]

The \textit{compound} goal \( p(X,Z), q(X,Y), r(Y,Z) \) can be executed without incurring a suspension if it is called with \( X \) ground.

The problem is to infer such a non-suspension property for the compound goal given \( d_i \) and \( g_i \) which describe non-suspension requirements and the success patterns for the atomic sub-goals.
Inferring a non-suspension requirement, $f$ say, for the compound goal from the $d_i$ and $g_i$:

**Proposition**
- Let $g_i \in \text{Bool}_X$ and $d_i \in \text{Mon}_X$ for all $i \in [1, m]$.
- Let $f \in \text{Def}_X$ where $f \models d = (\land_{i=1}^m (d_i \rightarrow g_i)) \rightarrow (\land_{j=1}^m d_j)$.
- Then there exists $i \in [1, m]$ such that $f \models d_i$.

We are interested in $m = 3$ and $X = \{X, Y, Z\}$. Moreover:

\[\land_{i=1}^3 (d_i \rightarrow g_i) = (X \rightarrow Z) \land (Z \rightarrow Y) \land_{i=1}^3 d_i = X \land Y \land Z \quad d = \ldots\]

Any $f \in \text{Def}_X$ such that $f \models d$ describes a state under which the compound goal can be executed without suspension.

To illustrate, consider $f = X = X \leftarrow \text{true} \in \text{Def}_X$.

Observe that $f \land \land_{i=1}^3 (d_i \rightarrow g_i) \models X \land Y \land Z \models (\land_{i=1}^3 d_i)$.

Hence $f \models \land_{i=1}^3 (d_i \rightarrow g_i) \rightarrow (\land_{i=1}^3 d_i)$ and indeed $f = X = d_1$. 
Non-suspension of the remaining sub-goals

The state after $p(X, Z)$ is described by $f \land g_1 = X \land Z \in \text{Def}_X$.

- Recall $f \models \land_{i=1}^3 (d_i \rightarrow g_i) \rightarrow (\land_{i=1}^3 d_i)$.
- Hence $f \land \land_{i=1}^3 (d_i \rightarrow g_i) \models (\land_{i=1}^3 d_i)$.
- Since $f \land g_1 \models f$, it follows $(f \land g_1) \land \land_{i=1}^3 (d_i \rightarrow g_i) \models f \land \land_{i=1}^3 (d_i \rightarrow g_i) \models (\land_{i=1}^3 d_i)$.
- Moreover $g_1 \models (d_1 \rightarrow g_1)$, thus $(f \land g_1) \land \land_{i=2}^3 (d_i \rightarrow g_i) \models (\land_{i=1}^3 d_i)$.
- But $(\land_{i=1}^3 d_i) \models (\land_{i=2}^3 d_i)$, hence $(f \land g_1) \land \land_{i=2}^3 (d_i \rightarrow g_i) \models (\land_{i=2}^3 d_i)$.
- Therefore $(f \land g_1) \models \land_{i=2}^3 (d_i \rightarrow g_i) \rightarrow (\land_{i=2}^3 d_i)$.

Reapplying the proposition, there must exist $i \in [2, 3]$ such that $f \land g_1 \models d_i$. Indeed $f \land g_1 \models X \land Z \models Z = d_3$, hence the third sub-goal can be executed without suspension.
Conclusions

- Backward analysis leads to a lightweight point-and-click approach to (partial) verification;
- Monotonic reordering results ensures scalability;
- The domain of boolean functions in rich enough to locate suspension bugs in real programs;
- Speed very significant in finding the needle in the haystack.