Exploring Interpolants

Philipp Rümmer, Pavle Subotić

Uppsala University, Sweden

COST Meeting, October 17
Introduction

Interpolants in Model Checking

- Craig interpolants used in model checking to refine abstractions
## Introduction

### Interpolants in Model Checking

- **Craig interpolants** used in model checking to refine abstractions
- For a given interpolation problem several interpolants may exist
Introduction

Interpolants in Model Checking

- Craig interpolants used in model checking to refine abstractions
- For a given interpolation problem several interpolants may exist
- The choice of interpolants affect if/how a program is verified
Introduction

Interpolants in Model Checking

- **Craig interpolants** used in model checking to refine abstractions
- For a given interpolation problem several interpolants may exist
- The **choice** of interpolants affect if/how a program is verified
- We present a technique that:
  - Discovers a *range* of interpolants
## Introduction

### Interpolants in Model Checking

- **Craig interpolants** used in model checking to refine abstractions
- For a given interpolation problem several interpolants may exist
- The **choice** of interpolants affect if/how a program is verified
- We present a technique that:
  - Discovers a *range* of interpolants
  - Incorporates *domain specific knowledge*
Introduction

Interpolants in Model Checking

- **Craig interpolants** used in model checking to refine abstractions
- For a given interpolation problem several interpolants may exist
- The **choice** of interpolants affect if/how a program is verified
- We present a technique that:
  - Discovers a *range* of interpolants
  - Incorporates *domain specific knowledge*
  - *Semantic* in nature
Introduction

Interpolants in Model Checking

- **Craig interpolants** used in model checking to refine abstractions
- For a given interpolation problem several interpolants may exist
- The **choice** of interpolants affect if/how a program is verified
- We present a technique that:
  - Discovers a *range* of interpolants
  - Incorporates *domain specific knowledge*
  - *Semantic* in nature
  - Prover independent
Preliminaries

Craig Interpolants

Let \((A \land B = false)\) then there exists an interpolant \(I\) for \((A, B)\) such that:

\[ A \rightarrow I \]
\[ B \rightarrow \neg I \]

\(I\) refers only to common symbols of \(A, B\)
Motivation

Motivating Example

```java
i = 0; x = j;     // init
while (i<50) {    // loop
    i++;
    x++;
}
if (j == 0)
    assert (x >= 50); // error location
```

Safety Properties

No feasible path exists that reaches an error state
Motivation

Analysis using CEGAR

1. Compute an approximation of CFG with respect to a set of predicates
Motivation

Analysis using CEGAR

1. Compute an approximation of CFG with respect to a set of predicates
2. Choose a (spurious or genuine) path to error
Motivation

Analysis using CEGAR

1. Compute an approximation of CFG with respect to a set of predicates
2. Choose a (spurious or genuine) path to error
3. If spurious, use interpolation to generate further predicates
Motivation

Motivating Example

```plaintext
i = 0; x = j;      // init
while (i<50) {
    i++;
    x++;
}
if (j == 0)
    assert (x >= 50); // error location
```

Counter Example - one loop iteration

\[
i_0 = 0 \land x_0 = j
\]
Motivation

Motivating Example

```
i = 0; x = j;  // init
while (i<50) {  // loop
    i++;
    x++;
}
if (j == 0)
    assert (x >= 50);  // error location
```

Counter Example - one loop iteration

```
i_0 = 0 \land x_0 = j \land i_0 < 50 \land i_1 = i_0 + 1 \land x_1 = x_0 + 1
```
Motivation

Motivating Example

```plaintext
i = 0; x = j; // init
while (i<50) { // loop
    i++;
    x++;
}
if (j == 0)
    assert (x >= 50); // error location
```

Counter Example - one loop iteration

Init

\[
i_0 = 0 \land x_0 = j \land i_0 < 50 \land i_1 = i_0 + 1 \land x_1 = x_0 + 1 \land i_1 \geq 50 \land j = 0 \land x_1 < 50
\]
Motivation

Counter Example - one loop iteration

\[ i_0 = 0 \land x_0 = j \land i_0 < 50 \land i_1 = i_0 + 1 \land x_1 = x_0 + 1 \land i_1 \geq 50 \land j = 0 \land x_1 < 50 \]

\( A \)

\( B \)

Interpolation Problem

\[ i_0 = 0 \land x_0 = j \land i_0 < 50 \land i_1 = i_0 + 1 \land x_1 = x_0 + 1 \rightarrow I \]

\( A \)

\( i_1 \geq 50 \land j = 0 \land x_1 < 50 \rightarrow \neg I \)

\( B \)

where \( I \) has symbols only from \( A \) and \( B \)
Motivation

Candidate Interpolant

\[ l_1 = (i_1 \leq 1) \]

The Interpolant

\[ i_0 = 0 \land x_0 = j \land i_0 < 50 \land i_1 = i_0 + 1 \land x_1 = x_0 + 1 \rightarrow i_1 \leq 1 \checkmark \]

\[ i_1 \geq 50 \land j = 0 \land x_1 < 50 \rightarrow \neg i_1 \leq 1 \checkmark \]

\[ i_1 \in \text{sym}(A) \text{ and } i_1 \in \text{sym}(B) \checkmark \]
Motivation

The Problem

- \((i_1 \leq 1)\) eliminates the counter-example
- Results in unrolling the loop - not \textit{general} enough
- What we really would like is an \textit{inductive invariant}
Motivation

A Better Candidate Interpolant

\[ l_2 = (x_1 \geq i_1 + j) \]

The Interpolant

\[
\begin{align*}
\underbrace{i_0 = 0 \land x_0 = j \land i_0 < 50 \land i_1 = i_0 + 1 \land x_1 = x_0 + 1} & \rightarrow (x_1 \geq i_1 + j) \checkmark \\
\underbrace{i_1 \geq 50 \land j = 0 \land x_1 < 50} & \rightarrow \neg(x_1 \geq i_1 + j) \checkmark \\
\end{align*}
\]

\[ x_1, i_1, j \in \text{sym}(A) \text{ and } x_1, i_1, j \in \text{sym}(B) \checkmark \]
Motivation

Interpolants

- \((x_1 \geq i_1 + j)\) avoids loop unrolling
- But how do we get \((x_1 \geq i_1 + j)\) instead of \((i_1 \leq 1)\) from the theorem prover?
Interpolant lattice for the example

\[
\begin{align*}
j \neq 0 \lor i_1 \leq 49 \lor x_1 \geq 50 & \quad I_T \\
i_1 \leq 49 & \\
i_1 \leq 2 & \\
i_1 \leq 1 & \quad I_1 \\
i_1 = 1 & \\
x_1 = j + 1 \land i_1 = 1 & \quad I_\bot \\
x_1 \geq i_1 + j & \\
x_1 = i_1 + j & \quad I_2 \\
j \neq 0 \lor x_1 \geq i_1 & \\
x_1 \geq i_1 + j & \\
x_1 = i_1 + j &
\end{align*}
\]
Interpolant lattice for the example

\[ j \neq 0 \lor i_1 \leq 49 \lor x_1 \geq 50 \]

- \( i_1 \leq 49 \)
  - \( i_1 \leq 2 \)
    - \( i_1 \leq 1 \)
      - \( i_1 = 1 \)
  - \( j \neq 0 \lor x_1 \geq i_1 \)
    - \( x_1 \geq i_1 + j \)
      - \( x_1 = i_1 + j \)
    - \( x_1 = j + 1 \land i_1 = 1 \)

- How to navigate in lattice?
- How to compare “quality” of interpolants?
Some Related Work

- **Syntactic restrictions** (R. Jhala and K. L. McMillan, TACAS 06)
- **Interpolant strength** (V. D’Silva VMCAI 10)
- **Beautiful Interpolants** (A. Albarghouthi, K. L. McMillan, CAV 13)
- **Term abstraction** (F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina, LPAR 12)
Our Approach

Pre-process the *interpolation query*
Our Approach

Pre-process the *interpolation query*

- General, prover independent framework
Our Approach

Pre-process the *interpolation query*

- General, prover independent framework
- Generate several interpolants for a given interpolation problem
Our Approach

Pre-process the *interpolation query*

- General, prover independent framework
- Generate several interpolants for a given interpolation problem
- Incorporate domain specific knowledge in defining interpolant quality
Outline

1. Interpolation Abstractions
2. Exploring Interpolants
3. Experiments on Software Programs
4. Conclusion
Abstractions in the Example

Step 1: Rename common variables in $A[\bar{s}_A, \bar{s}] \land B[\bar{s}, \bar{s}_B]$.

In the example: common symbols are $\{j, i_1, x_1\}$.

\[
A[\bar{s}_A, \bar{s}'] = i_0 = 0 \land x_0 = j' \land i_0 < 50 \land i'_1 = i_0 \land x'_1 = x_0
\]

\[
B[\bar{s}'', \bar{s}_B] = i''_1 \geq 50 \land j'' = 0 \land x''_1 < 50
\]
Abstractions in the Example

- Step 1: Rename common symbols in $A[\bar{s}_A, \bar{s}] \land B[\bar{s}, \bar{s}_B]$
- Step 2: Add templates capturing limited knowledge

In the example: templates are $\{j, x_1 - i_1\}$

\[
A[\bar{s}_A, \bar{s}] = i_0 = 0 \land x_0 = j' \land i_0 < 50 \land i'_1 = i_0 \land x'_1 = x_0 \land x'_1 - i'_1 = x_1 - i_1 \land j' = j \land R_A[\bar{s}', \bar{s}]
\]

\[
B[\bar{s}, \bar{s}_B] = i''_1 \geq 50 \land j'' = 0 \land x''_1 < 50 \land x_1 - i_1 = x''_1 - i''_1 \land j = j'' \land R_B[\bar{s}, \bar{s}'']
\]
Example

Interpolation Problem $A \land B$
Example

With abstraction generated by template $x - y$
Example

Blocks Interpolants $x \geq 4$ etc.
Example

Allows interpolants $x \geq y$ etc.
Interpolant sub-lattice for templates \( \{i_1\} \) and \( \{j, x_1 - i_1\} \)

\[
\begin{align*}
    j \neq 0 & \lor i_1 \leq 49 \lor x_1 \geq 50 \\
    i_1 \leq 49 & \\
    i_1 \leq 2 & \\
    i_1 \leq 1 & \quad I_1 \\
    i_1 = 1 & \\
    x_1 = i_1 + j & \quad I_2 \\
    x_1 \geq i_1 + j & \\
    x_1 = j + 1 & \land i_1 = 1 \quad I_\bot
\end{align*}
\]
Definitions

Definition (Abstraction)

An interpolation abstraction is a pair \((R_A[\bar{s}', \bar{s}], R_B[\bar{s}, \bar{s}''])\) of formulae with the property that \(R_A[\bar{s}, \bar{s}]\) and \(R_B[\bar{s}, \bar{s}]\) are valid i.e., \(Id[\bar{s}', \bar{s}] \Rightarrow R_A[\bar{s}', \bar{s}]\) and \(Id[\bar{s}, \bar{s}''] \Rightarrow R_B[\bar{s}, \bar{s}'']\).

Definition (Abstract Interpolation Problem)

- \(A[\bar{s}_A, \bar{s}] \land B[\bar{s}, \bar{s}_B]\) is the concrete interpolation problem.
- \((A[\bar{s}_A, \bar{s}'] \land R_A[\bar{s}, \bar{s}']) \land (R_B[\bar{s}'', \bar{s}] \land B[\bar{s}'', \bar{s}_B])\) is called abstract interpolation problem;

Definition (Feasible Abstractions)

Assuming that the concrete interpolation problem is solvable, we call an interpolation abstraction feasible if also the abstract interpolation problem is solvable, and infeasible otherwise.
Natural classes of Abstractions

- **Term interpolation abstractions**, constructed from a set of terms \( \{t_1, t_2, \ldots, t_n\} \)

  \[
  R^T_A[\vec{s}', \vec{s}] = \bigwedge_{i=1}^{n} t_i[\vec{s}'] = t_i[\vec{s}], \quad R^T_B[\vec{s}, \vec{s}''] = \bigwedge_{i=1}^{n} t_i[\vec{s}] = t_i[\vec{s}'']
  \]

  (same possible for inequalities)

- **Predicate interpolation abstractions**, constructed from \( \{\phi_1, \phi_2, \ldots, \phi_n\} \)

  \[
  R^{Pred}_A[\vec{s}', \vec{s}] = \bigwedge_{i=1}^{n} (\phi_i[\vec{s}'] \rightarrow \phi_i[\vec{s}]), \quad R^{Pred}_B[\vec{s}, \vec{s}'''] = \bigwedge_{i=1}^{n} (\phi_i[\vec{s}] \rightarrow \phi_i[\vec{s}'''])
  \]

- Quantified interpolation abstractions
  - …
Soundness and Completeness

Lemma (Soundness)

Every interpolant of the abstract interpolation problem is also an interpolant of the concrete interpolation problem (but in general not vice versa).

Lemma (Completeness)

Suppose $A[\bar{s}_A, \bar{s}] \land B[\bar{s}, \bar{s}_B]$ is an interpolation problem with interpolant $I[\bar{s}]$, such that both $A[\bar{s}_A, \bar{s}]$ and $B[\bar{s}, \bar{s}_B]$ are satisfiable. Then there is a feasible interpolation abstraction such that every abstract interpolant is equivalent to $I[\bar{s}]$. 
How do we find good interpolation abstractions?
Can be done in two steps:

- Define a base vocabulary of “interesting” templates (building blocks for interpolants)
- Search for **maximum feasible** interpolation abstractions in this language
How do we find good interpolation abstractions?

Can be done in two steps:

- Define a base vocabulary of “interesting” templates (building blocks for interpolants)
- Search for **maximum feasible** interpolation abstractions in this language

**Definition (Abstraction lattice)**

Suppose an interpolation problem $A[\bar{s}_A, \bar{s}] \land B[\bar{s}, \bar{s}_B]$. An **abstraction lattice** is a pair $(\langle L, \sqsubseteq_L \rangle, \mu)$ consisting of a complete lattice $\langle L, \sqsubseteq_L \rangle$ and a monotonic mapping $\mu$ from elements of $\langle L, \sqsubseteq_L \rangle$ to interpolation abstractions $(R_A[\bar{s}', \bar{s}], R_B[\bar{s}, \bar{s}''])$ with the property that $\mu(\perp) = (Id[\bar{s}', \bar{s}], Id[\bar{s}, \bar{s}''])$. 
Abstraction lattice template base set \( \{ x_1 - i_1, i_1, j \} \)
Sub-lattices of interpolant lattice

\[ j \neq 0 \vee i_1 \leq 49 \vee x_1 \geq 50 \]

\[ i_1 \leq 49 \]

\[ i_1 \leq 2 \]

\[ i_1 \leq 1 \]

\[ i_1 = 1 \]

\[ j \neq 0 \vee x_1 \geq i_1 \]

\[ x_1 \geq i_1 + j \]

\[ x_1 = i_1 + j \]

\[ x_1 = j + 1 \land i_1 = 1 \]
Overall Architecture
Overall Architecture

Exploration of Abstraction Lattice

Verifier

Interpolation Engine

Interpolation Abstraction

Light-weight static analysis

Predicates

CEGAR loop

Abstract query(s)

Feasible abstractions

Counter example

Template lattice

Domain knowledge
Experiments

Experiment Setup

- Extended the Eldarica model checker with our approach
- Experiments on Horn clause benchmarks generated from programs
- Pre-computed templates of the form \( \{x, y, x - y, x + y\} \)
  Typically 15–300 templates
- Costs assigned to templates to define preference
## Experiments

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Eldarica</th>
<th>Eldarica-ABS</th>
<th>Flata</th>
<th>Z3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N</td>
<td>sec</td>
<td>N</td>
<td>sec</td>
</tr>
<tr>
<td><strong>C programs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>boustrophedon (C)</td>
<td>*</td>
<td>*</td>
<td>10</td>
<td>10.7</td>
</tr>
<tr>
<td>boustrophedon_expanded (C)</td>
<td>*</td>
<td>*</td>
<td>11</td>
<td>7.7</td>
</tr>
<tr>
<td>halbwachs (C)</td>
<td>*</td>
<td>*</td>
<td>53</td>
<td>2.4</td>
</tr>
<tr>
<td>gopan (C)</td>
<td>17</td>
<td>22.2</td>
<td>62</td>
<td>57.0</td>
</tr>
<tr>
<td>rate_limiter (C)</td>
<td>11</td>
<td>2.7</td>
<td>11</td>
<td>19.1</td>
</tr>
<tr>
<td>anubhav (C)</td>
<td>1</td>
<td>1.7</td>
<td>1</td>
<td>1.6</td>
</tr>
<tr>
<td>cousot (C)</td>
<td>*</td>
<td>*</td>
<td>3</td>
<td>7.7</td>
</tr>
<tr>
<td>bubblesort (E)</td>
<td>1</td>
<td>2.8</td>
<td>1</td>
<td>2.3</td>
</tr>
<tr>
<td>insdel (C)</td>
<td>1</td>
<td>0.9</td>
<td>1</td>
<td>0.9</td>
</tr>
<tr>
<td>insertsort (E)</td>
<td>1</td>
<td>1.8</td>
<td>1</td>
<td>1.7</td>
</tr>
<tr>
<td>listcounter (C)</td>
<td>*</td>
<td>*</td>
<td>8</td>
<td>2.0</td>
</tr>
<tr>
<td>listcounter (E)</td>
<td>1</td>
<td>0.9</td>
<td>1</td>
<td>0.9</td>
</tr>
<tr>
<td>listreversal (C)</td>
<td>1</td>
<td>1.9</td>
<td>1</td>
<td>1.9</td>
</tr>
<tr>
<td>mergesort (E)</td>
<td>1</td>
<td>2.9</td>
<td>1</td>
<td>2.6</td>
</tr>
<tr>
<td>selectionsort (E)</td>
<td>1</td>
<td>2.4</td>
<td>1</td>
<td>2.4</td>
</tr>
<tr>
<td>rotation_vc.1 (C)</td>
<td>7</td>
<td>2.0</td>
<td>7</td>
<td>0.3</td>
</tr>
<tr>
<td>rotation_vc.2 (C)</td>
<td>8</td>
<td>2.7</td>
<td>8</td>
<td>0.2</td>
</tr>
<tr>
<td>rotation_vc.3 (C)</td>
<td>0</td>
<td>2.3</td>
<td>0</td>
<td>2.3</td>
</tr>
<tr>
<td>rotation.1 (E)</td>
<td>3</td>
<td>1.8</td>
<td>3</td>
<td>1.8</td>
</tr>
<tr>
<td>split_vc.1 (C)</td>
<td>18</td>
<td>3.9</td>
<td>17</td>
<td>3.2</td>
</tr>
<tr>
<td>split_vc.2 (C)</td>
<td>*</td>
<td>*</td>
<td>18</td>
<td>1.1</td>
</tr>
<tr>
<td>split_vc.3 (C)</td>
<td>0</td>
<td>2.8</td>
<td>0</td>
<td>1.5</td>
</tr>
<tr>
<td><strong>Recursive Horn SMT-LIB Benchmarks</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addition (C)</td>
<td>1</td>
<td>0.7</td>
<td>1</td>
<td>0.8</td>
</tr>
<tr>
<td>bfprt (C)</td>
<td>*</td>
<td>*</td>
<td>5</td>
<td>8.3</td>
</tr>
<tr>
<td>binarysearch (C)</td>
<td>1</td>
<td>0.9</td>
<td>1</td>
<td>0.9</td>
</tr>
<tr>
<td>buildsheap (C)</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>countZero (C)</td>
<td>2</td>
<td>2.0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td>disjunctive (C)</td>
<td>10</td>
<td>2.4</td>
<td>5</td>
<td>5.0</td>
</tr>
<tr>
<td>floodfill (C)</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>gcd (C)</td>
<td>4</td>
<td>1.2</td>
<td>4</td>
<td>2.0</td>
</tr>
<tr>
<td>identity (C)</td>
<td>2</td>
<td>1.1</td>
<td>2</td>
<td>1.1</td>
</tr>
<tr>
<td>merge-leq (C)</td>
<td>3</td>
<td>1.1</td>
<td>7</td>
<td>1.1</td>
</tr>
</tbody>
</table>
Summary

A semantic, solver-independent framework for guiding interpolant search
Summary

A semantic, solver-independent framework for guiding interpolant search

- We pre-process the interpolation queries
Summary

A semantic, solver-independent framework for guiding interpolant search

- We pre-process the interpolation queries
  - Easy to integrate in verifiers (basic implementation 500-1000 LOC)
A semantic, solver-independent framework for guiding interpolant search

- We pre-process the interpolation queries
  - Easy to integrate in verifiers (basic implementation 500-1000 LOC)
  - Enables use of domain-specific knowledge in interpolation
Summary

A semantic, solver-independent framework for guiding interpolant search

- We pre-process the interpolation queries
  - Easy to integrate in verifiers (basic implementation 500-1000 LOC)
  - Enables use of domain-specific knowledge in interpolation
- General framework
A semantic, solver-independent framework for guiding interpolant search

- We pre-process the interpolation queries
  - Easy to integrate in verifiers (basic implementation 500-1000 LOC)
  - Enables use of domain-specific knowledge in interpolation
- General framework
  - Our implementation is just a basic instance of the framework
Summary

A semantic, solver-independent framework for guiding interpolant search

- We pre-process the interpolation queries
  - Easy to integrate in verifiers (basic implementation 500-1000 LOC)
  - Enables use of domain-specific knowledge in interpolation

- General framework
  - Our implementation is just a basic instance of the framework
  - Each query can have a specific lattice, lattices can be infinite etc.
Summary

A semantic, solver-independent framework for guiding interpolant search

- We pre-process the interpolation queries
  - Easy to integrate in verifiers (basic implementation 500-1000 LOC)
  - Enables use of domain-specific knowledge in interpolation

- General framework
  - Our implementation is just a basic instance of the framework
  - Each query can have a specific lattice, lattices can be infinite etc.
  - Applicable to various logics, not restricted to arithmetic
Summary

A semantic, solver-independent framework for guiding interpolant search

- We pre-process the interpolation queries
  - Easy to integrate in verifiers (basic implementation 500-1000 LOC)
  - Enables use of domain-specific knowledge in interpolation

- General framework
  - Our implementation is just a basic instance of the framework
  - Each query can have a specific lattice, lattices can be infinite etc.
  - Applicable to various logics, not restricted to arithmetic

- Templates, but interpolants still constructed by theorem prover
  \[\Rightarrow\] Arbitrary Boolean structure, etc., allowed
Summary

Applications (ongoing work)

- Software programs with heap, other datatypes
- Timed systems
- Reachability in Petri nets/Vector addition systems
Thank you - Questions
Finding Abstractions

Algorithm 1: Exploration algorithm

**Input:** Interpolation problem $A[\bar{s}_A, \bar{s}] \land B[\bar{s}, \bar{s}_B]$, abstraction lattice $(\langle L, \sqsubseteq_L \rangle, \mu)$

**Result:** Set of maximal feasible interpolation abstractions

1. if $\bot$ is infeasible then
2.   return $\emptyset$;
3. end

4. $\text{Frontier} \leftarrow \{ \text{maximise}(\bot) \}$;

5. while $\exists$ feasible $\text{elem} \in L$, incomparable with $\text{Frontier}$ do
6.   $\text{Frontier} \leftarrow \text{Frontier} \cup \{ \text{maximise}(\text{elem}) \}$;
7. end
8. return $\text{Frontier}$;
Finding Abstractions

Algorithm 2: Maximisation algorithm

Input: Feasible element: \( elem \)
Result: Maximal feasible element

while \( \exists \) feasible successor \( fs \) of \( elem \) do
    pick element \( middle \) such that \( fs \sqsubseteq_i L \middle\uparrow middle \sqsubseteq_i T \);
    if \( middle \) is feasible then
        \( elem \leftarrow middle \);
    else
        \( elem \leftarrow fs \);
    end
end
return \( elem \);