Extending Dynamic Constraint Detection with Disjunctive Constraints

Nadya Kuzmina
John Paul
Ruben Gamboa
James Caldwell

University of Wyoming
Dynamic Constraint Detection

- Fixed grammar of universal properties.
  - Serves well for the discovery of a well-defined set of problem-specific, but program-independent properties.
  - Does not allow to capture the logic of a particular program.

- Goal: enable constraint detection to capture the subtle essential properties of a program under analysis.
State Space Partitioning Technique (SSPT)

- Combines static and dynamic program analysis.
- Automatically specializes the language of constraint detection.
- Adds program-specific disjunctive properties.
Introduction: State Space Partitions

if \(x < 0\) \{\ldots\} \quad \Rightarrow \quad P_1 \equiv x < 0,
else if \(y > 0\) \{\ldots\} \quad \Rightarrow \quad P_2 \equiv x \geq 0 \land y > 0,
else \{\ldots\} \quad \Rightarrow \quad P_3 \equiv x \geq 0 \land y \leq 0

State space: \(\{\langle x, y \rangle \mid -2^{31} \leq x, y < 2^{31}\}\)

Three disjoint subspaces, or abstract states: \(P_1, P_2, P_3\)
Types of Disjunctive Constraints

- **Object Invariant**
  - Properties $a$ and $b$ are mutually exclusive: $\neg a \lor \neg b$

- **Use cases for a method $m$**
  - Method $m$ was called when abstract states $s$ or $w$ hold: $s \lor w$

- **Transitions between abstract states induced by a method $m$**
  - $p \Rightarrow q$
  - $p$ is an abstract state on variables at precondition of $m$
  - $q$ is a disjunction of abstract states on variables at postcondition of $m$

- **Daikon-inferred implications for a method $m$**
  - $p \Rightarrow t$
  - $p$ is an abstract state on variables at precondition of $m$
  - $t$ is an instantiated template
public class CalcEngine {

    //number which appears in the Calculator display
    private int displayValue;
    //store a running total
    private int total;
    //true if #’s pressed should overwrite display
    private boolean newNumber;
    //true if adding
    private boolean adding;
    //true if subtracting
    private boolean subtracting;

    public void numberPressed(int number) {
        if (newNumber)
            displayValue = number;
        else
            displayValue = displayValue * 10 + number;
        newNumber = false;
    }

    public void equals() {
        if (adding)
            displayValue = displayValue + total;
        else if (subtracting)
            displayValue = total - displayValue;
        ...
    }

    public void clear() { ...

    public void plus() { ...

    public void minus() { ...

}
State Spaces for the Calculator Example

```java
public class CalcEngine {
    //number which appears in the Calculator display
    private int displayValue;
    //store a running total
    private int total;
    //true if #’s pressed should overwrite display
    private boolean newNumber;
    //true if adding
    private boolean adding;
    //true if subtracting
    private boolean subtracting;

    public void numberPressed(int number) {
        if (newNumber)
            displayValue = number;
        else
            displayValue = displayValue * 10 + number;
        newNumber = false;
    }

    public void equals() {
        if (adding)
            displayValue = displayValue + total;
        else if (subtracting)
            displayValue = total - displayValue;
        ...
    }

    public void clear() {
    }

    public void plus() {
    }

    public void minus() {
    }
}
```

\[ \Pi_1 \equiv \{ P_1, P_2 \} \]

\[ P_1 \equiv \text{newNumber} \]

\[ P_2 \equiv \neg \text{newNumber} \]

\[ \Pi_2 \equiv \{ Q_1, Q_2, Q_3 \} \]

\[ Q_1 \equiv \text{adding} \]

\[ Q_2 \equiv \neg \text{adding} \land \neg \text{subtracting} \]

\[ Q_3 \equiv \neg \text{adding} \land \neg \text{subtracting} \]
Constraints for Calculator

```java
public class CalcEngine {

    // number which appears in the Calculator display
    private int displayValue;
    // store a running total
    private int total;
    // true if #'s pressed should overwrite display
    private boolean newNumber;
    // true if adding
    private boolean adding;
    // true if subtracting
    private boolean subtracting;

    public void numberPressed(int number) {
        if (newNumber)
            displayValue = number;
        else
            displayValue = displayValue * 10 + number;
        newNumber = false;
    }

    public void equals() {
        if (adding)
            displayValue = displayValue + total;
        else if (subtracting)
            displayValue = total - displayValue;
        ...
    }

    public void clear() { ...
    }

    public void plus() { ...
    }

    public void minus() { ...
    }
}

\Pi_1 \equiv \{P_1, P_2\}
\Pi_2 \equiv \{Q_1, Q_2, Q_3\}
\newNumber \equiv P_1
\neg \newNumber \equiv P_2
Q_1 \equiv \text{adding}
Q_2 \equiv \neg \text{adding} \land \neg \text{subtracting}
Q_3 \equiv \neg \text{adding} \land \neg \text{subtracting}

Object Invariant:
context CalcEngine inv:
(!this.adding || !this.subtracting)

Method Constraints:
context CalcEngine::numberPressed(int number)
pre: P1 || P2, Q1 || Q2 || Q3
post: orig(P1) ==> P2, orig(P2) ==> P2
     orig(Q1) ==> Q1, orig(Q2) ==> Q2
     orig(Q3) ==> Q3
     orig(P1) <==> (displayValue == orig(number))
     orig(P2) <==> 
        (displayValue ==
         10 \times \text{orig(displayValue)} + \text{orig(number)})
context CalcEngine::clear()
pre: P1 || P2, Q3
post: orig(P1) ==> P1, orig(P2) ==> P1,
     orig(Q3) ==> Q3
```
SSPT: Overview

- Form disjoint partitions of the state spaces of the program variables involved in expressing the if-then-else tests.

\[
\begin{align*}
\text{if (adding)} \\
... \\
\text{else if (subtracting)} \\
... \\
\Pi_2 &\equiv \{Q_1, Q_2, Q_3\} \\
Q_1 &\equiv \text{adding} \\
Q_2 &\equiv \neg\text{adding} \land \neg\text{subtracting} \\
Q_3 &\equiv \neg\text{adding} \land \neg\text{subtracting}
\end{align*}
\]
SSPT: Hypothesized Constraints

Let $\Pi=\{P_1, P_2, \ldots, P_n\}$

- Preconditions: $P_1 \lor P_2 \lor \ldots \lor P_n$
- Postconditions: $P_i \Rightarrow P_j \lor P_k$, $i, j, k \in [1..n]$
- Object invariants: check whether the tests of the corresponding if-then-else statement are mutually exclusive.
  - For the Calculator example

  
  $\begin{align*}
  \text{if (adding)} & \quad (\text{adding} \land \neg \text{subtracting}) \lor \\
  \quad \ldots & \quad (\neg \text{adding} \land \text{subtracting}) \lor \\
  \text{else if (subtracting)} & \quad (\neg \text{adding} \land \neg \text{subtracting})
  \end{align*}$
SSPT: Constraint Approximation Algorithm

- Let $\Pi = \{P_1, P_2, P_3\}$
- Notation: for $i \in [1..3]$
  - $P_i^{pre}$ - abstract state $P_i$ over variable values at precondition
  - $P_i^{post}$ - abstract state $P_i$ over variable values at postcondition
SSPT: Constraint Approximation Algorithm

At the post-condition program point for a method $M$ compute the transitional post-condition for each $P_i^{pre}$, $i \in [1..3]$, as follows:

1. Assume that $P_i^{pre} \Rightarrow \neg P_1^{post}$, $P_i^{pre} \Rightarrow \neg P_2^{post}$, and $P_i^{pre} \Rightarrow \neg P_3^{post}$ are all possible transitions. Denote this by the set $S$ of indices $S = \{1, 2, 3\}$.

2. Perform dynamic analysis, and whenever $P_i^{pre}$ and $P_j^{post}$ both hold, remove $j$ from $S$.

3. Approximate the transitional post-condition for $P_i^{pre}$ with a disjunction of abstract states whose indices are contained in the complement of $S$, $P_i^{pre} \Rightarrow \bigvee_{k \in S^c} P_k^{post}$.
SSPT: Constraint Approximation Algorithm

Intuition behind the algorithm:
Let \( i = 1 \) and after step 2, let \( S = \{1, 3\} \).
Then, \( P_1^{\text{pre}} \Rightarrow \neg P_1^{\text{post}} \) and \( P_1^{\text{pre}} \Rightarrow \neg P_3^{\text{post}} \) are consistent with the observed data.

\( P_1^{\text{post}} \lor P_2^{\text{post}} \lor P_3^{\text{post}} \) is true by construction.

The transition \( P_1^{\text{pre}} \Rightarrow P_2^{\text{post}} \) follows by propositional logic.
ContExt: Implementation

- Lightweight static analysis of Java source code for abstract state extraction.
- Dynamic analysis tasks are delegated to Daikon.
- ContExt combines the constraints inferred by our approach with those inferred by Daikon in its output.
Transitional Constraint Inference

- A splitting condition (splitter) is a boolean expression in terms of some program variables.
- Let \( T \) be a program point which has all the variables involved in a splitter \( a \).
- \( a \) partitions the data trace into two mutually exclusive subsets:
  - \( T_a \) : contains the data values that satisfy \( a \)
  - \( T_{\neg a} \) : contains the data values on which \( a \) does not hold.
- Each abstract state from a space is used as a splitter on the data trace at postcondition program points of the enclosing class.
- Convenient checks when \( \Pi_{P_i^{pre}} \) and \( \Pi_{P_j^{post}} \) both hold.
Limitations

- Our approach is primarily a dynamic analysis.
  - The reported constraints are unsound.
  - Potentially stronger constraints are reported.
- Increase in the number of accidental constraints reported and loss of precision.
- Given the same test suite, our approach may not infer some unconditional constraints that Daikon would.
- Requires the presence of source code.
- The technique has been applied to only one class at a time.
Evaluation Challenge

- Quantitative measurement of the quality of inferred constraints is challenging.
- Propose a methodology for a quantitative evaluation of constraint inference techniques based on a modeling language Alloy.
- Concentrate on recall.
- Apply it to comparatively evaluate Daikon and ContExt on two examples.
Evaluation Methodology
Case Study 1: Puzzle

- The Puzzle class represents an environment with an agent.
Puzzle Specification

Assertion in Alloy

```alloy
assert moveForward_1 {
  all p’: Puzzle, p : Puzzle |
  (p in {p’.moveForward.Unit}) =>
  (p’.yPosition = p.yPosition <=> p’.yPosition = 0)
}
assert moveForward_2 {
  all p’: Puzzle, p : Puzzle |
  (p in {p’.moveForward.Unit}) =>
  (p’.yPosition - 1 = p.yPosition <=> p’.yPosition > 0)
}
assert moveForward_3 {
  all p’: Puzzle, p : Puzzle |
  (p in {p’.moveForward.Unit}) =>
  p.yPosition =< p’.yPosition
}
assert moveForward_4 {
  all p’: Puzzle, p : Puzzle |
  (p in {p’.moveForward.Unit}) =>
  (p.xPosition = p’.xPosition)
}
```

English-language specification

The y-coordinate of the agent is to remain the same only when it attempts a moveForward from the top edge of the board (y is 0).

Otherwise, an agent moves forward exactly one square (y-coordinate decreases by one).

The y-position of the agent at the post-condition of the moveForward method is less than or equal to the y-position at pre-condition.

Moving forward does not affect the x-coordinate of the agent.
# Puzzle Evaluation

<table>
<thead>
<tr>
<th></th>
<th>number of assertions</th>
<th>number of checked assertions</th>
<th>number of facts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Daikon</td>
<td>35</td>
<td>18 (51%)</td>
<td>35</td>
</tr>
<tr>
<td>Daikon (w/split)</td>
<td>35</td>
<td>23 (66%)</td>
<td>124</td>
</tr>
<tr>
<td>ContExt</td>
<td>35</td>
<td>28 (80%)</td>
<td>554</td>
</tr>
</tbody>
</table>

*Comparative evaluation of the inferred constraints for ContExt and Daikon on the Puzzle example*
## Case Study 2: Employee Example

<table>
<thead>
<tr>
<th></th>
<th>number of assertions</th>
<th>number of checked assertions</th>
<th>number of facts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Daikon</td>
<td>15</td>
<td>12 (80%)</td>
<td>55</td>
</tr>
<tr>
<td>ContExt</td>
<td>15</td>
<td>15 (100%)</td>
<td>89</td>
</tr>
</tbody>
</table>

**Comparative evaluation of the inferred constraints for ContExt and Daikon on the Employee example**
Related Work

- Csallner et al. employ a dynamic symbolic execution technique to obtain program-specific constraints.
  - performs symbolic execution over an existing test suite.
- Engler et al. and Yang et al. focus on recovering a relatively small number of error-revealing properties.
- Dallmaier et al. use a combination of static and dynamic analysis to construct state machines that represent an object’s behavior in terms of its inspector and mutator methods.
- Arumuga Nainar et al. are interested in finding relevant boolean formulae.
  - The formulae partition the program state space into only two subspaces, one in which a bug is exhibited, and the other one in which it is not.
Conclusions

- State Space Partitioning Technique combines lightweight static and dynamic analysis to provide for the inference of program-specific disjunctive properties.

- Proposed an evaluation methodology for the quality of inferred constraints based on the Alloy modeling language.
Comparative Complexity

- Generalized disjunctive template:
  - $2^k$, where $k$ is the number of hypothesized non-disjunctive constraints.
Comparative Complexity

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$</td>
<td>Number of program points in the target program.</td>
</tr>
<tr>
<td>$C$</td>
<td>Number of hypothesized constraints at a program point.</td>
</tr>
<tr>
<td>$L$</td>
<td>Number of data samples observed.</td>
</tr>
</tbody>
</table>

- **Daikon** *(approximated with those of the simple incremental algorithm)*
  - Space complexity: $S = O(P \times C)$
  - Time complexity: $T = O(P \times C \times L)$
Comparative Complexity

<table>
<thead>
<tr>
<th>$P$</th>
<th>Number of program points in the target program.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C$</td>
<td>Number of hypothesized constraints at a program point.</td>
</tr>
<tr>
<td>$L$</td>
<td>Number of data samples observed.</td>
</tr>
<tr>
<td>$m$</td>
<td>Number of class-scoped partitions.</td>
</tr>
<tr>
<td>$n$</td>
<td>The maximum number of states per class-scoped partition.</td>
</tr>
</tbody>
</table>

**ContExt:**

- $P' = m \times n \times P$, $C' = m \times n + C$
- **Space complexity:** $S = O(P' \times C') = O(mnP \times (mn + C))$
- **Time complexity:** $T = O(P' \times C' \times L) = O(mnP \times (mn + C) \times L)$