Alloy as a refactoring checker?

H.-Christian Estler, Heike Wehrheim

Universität Paderborn

Refine 2008 - International Refinement Workshop
Motivation

“As a program is evolved its complexity increases unless work is done to maintain or reduce it.” M. M. Lehman
Motivation

- **Refactorings** are systematic changes to improve the structure of a program, e.g.
  - Simplify operations
  - Improve reusability
  - Increase readability

- Used for programs but also models or specifications

- Important: refactorings must preserve the external observable behavior
Motivation

How to check behavior-preservation?
- Usual approach: testing
- Use template pairs (describing before and after state)
- Use an automatic verification tool

Subject of our work
- Can the Alloy Analyzer be used to verify behavior-preservation of refactorings for Z specifications?
Overview

1. Translating a Z specification into the Alloy language
2. Defining *behavior-preservation* for refactorings in Z
3. Applying the Alloy Analyzer for verification
What is Alloy?

- Alloy = Alloy language + Alloy Analyzer
- developed by the Software Design Group at MIT

Alloy language
- Declarative specification language (based on first order logic)
- Strongly inspired by Z

Alloy Analyzer
- SAT based constraint solver
- Automatic simulation and analysis of Alloy models
- A model finder: tries to find a model for a formula
Example of a translation

sig ELEMENT {}

sig Set {
    elements: set ELEMENT
}

pred Add_Elem[s, s': Set,
    e_in: ELEMENT]{
    e_in not in s.elements
    s'.elements = s.elements + e_in
}

/* run a simulation */
run {} for 3
Structure of an Alloy model

```
sig ELEMENT {}  

sig Set {
    elements: set ELEMENT
}

pred Add_ELEM[s, s': Set,
    e_in: ELEMENT] {
    e_in not in s.elements
    s'.elements = s.elements + e_in
}

/* run a simulation */
run {} for 3
```

- Signatures define the state space
- Model consists of *atoms* and *relations*
Translating Z into Alloy

Structure of an Alloy model

```alloy
sig ELEMENT {} 

sig Set { 
    elements: set ELEMENT 
}

pred Add_Elem[s, s': Set, 
    e_in: ELEMENT]{} 
    e_in not in s.elements 
    s'.elements = s.elements + e_in 
}

/* run a simulation */
run {} for 3
```

- Signatures define the state space
- Model consists of *atoms* and *relations*

![Diagram of Set, Element0, Element1, Element2 relationships]
Structure of an Alloy model

```alloy
sig ELEMENT {}

sig Set {
  elements: set ELEMENT
}

pred Add_Elem[s, s': Set, e_in: ELEMENT] {
  e_in not in s.elements
  s'.elements = s.elements + e_in
}

/* run a simulation */
run Add_Elem for 3
```

- Z operations are translated to predicates

![Diagram showing the translation of Z operations to Alloy]

Set0 (Add_Elem_s)
Set1 (Add_Elem_s')
Element0 (Add_Elem_e_in)
Element1
Checking properties of an Alloy model

- Use *assertions* to check properties of a model, e.g.

```alloy
/* Assertion: there are no empty sets */
assert EmptySet { all s: Set | #s.elements > 0 }

check EmptySet for 3 but 2 Set
```

- Alloy Analyzer examines every possible instance
Checking properties of an Alloy model

- Use assertions to check properties of a model, e.g.

```alloy
/* Assertion: there are no empty sets */
assert EmptySet { all s: Set | #s.elements > 0 }

check EmptySet for 3 but 2 Set
```

- Alloy Analyzer examines every possible instance
How to check refactorings?

- Remember: refactorings must not change the external behavior (behavior-preservation)
- Refinement guarantees substitutability
  - But might be irreversible
- Therefore, use refinement in “both directions”

Definition

Tow specifications $A$ and $C$ are behavior-preserving, iff $A \sqsubseteq C$ and $C \sqsubseteq A$. 
Checking Refinement using downward simulation

1. **Init:**
   \[ \forall CState' \bullet CInit \Rightarrow \exists AState' \bullet AInit \land R' \]

2. **Applicability:**
   \[ \forall AState; CState \bullet R \Rightarrow (\text{pre } COp_i \leftrightarrow \text{pre } AOp_i) \]

3. **Correctness:**
   \[ \forall AState; CState; CState' \bullet R \land COp_i \Rightarrow \\
   \exists AState' \bullet R' \land AOp_i \]
Translate conditions into Alloy assertions

- Alloy allows direct translation, e.g.

- *Correctness:*
  \[ \forall AState; CState; CState' \cdot R \land COp_i \Rightarrow \exists AState' \cdot R' \land AOp_i \]

```alloy
assert Correct {
  all a: AState, c,c': CState | R[a,c] and COp_i =>
    {some a': AState | R[a',c'] and AOp_i}
}
```
Translate conditions into Alloy assertions

- Alloy allows direct translation, e.g.

- **Correctness:**
  \[
  \forall AState; CState; CState' \bullet R \land COp_i \Rightarrow \\
  \exists AState' \bullet R' \land AOp_i
  \]

```alloy
assert Correct { 
  all a: AState, c,c': CState| R[a,c] and COp_i => 
  {some a': AState| R[a',c'] and AOp_i}
}
```

- But, verification will **fail** due to the use of $\exists$ in the consequence of an implication
Problem with existential quantification

```alloy
assert Closed {
  all s0, s1: Set | some s2: Set |
  s2.elements = s0.elements + s1.
    elements
}
```

- Analyzer negates assertion
- Tries to find model for the negation

```alloy
some s0, s1: Set | all s2: Set |
not s2.elements = s0.elements +
  s1.elements
```
Problem with existential quantification

```
assert Closed { all s0, s1: Set | some s2: Set | s2.elements = s0.elements + s1.elements }
```

```
some s0, s1: Set | all s2: Set | not s2.elements = s0.elements + s1.elements
```

- Analyzer negates assertion
- Tries to find model for the negation
- Problem: actual instance of the model can be too small
Solutions to this problem?

- Constrain the model to fully populate the state space (*generator axiom*).

\[
\text{fact} \begin{cases} 
\text{some } s: \text{Set} | \text{no } s.\text{elements} \\
\text{all } s: \text{Set}, e: \text{ELEMENT} | \text{some } s':\text{Set} | \\
\text{s'.elements} = s.\text{elements} + e 
\end{cases}
\]

- Analysis becomes intractable as scope explodes
  - To analyze \( n \) ELEMENT we need \( 2^n \) Set

- Instead: try to omit existential quantifier
Simplifying the refinement conditions

- A lot of refactorings do not change the state space
- Thus, representation relation $R$ is the identity

Given that $R$ is total and bijective:
$A \sqsubseteq_{DS} C$ and $C \sqsubseteq_{DS} A$ iff

1. **Init:**
   \[ \forall AState', CState' \bullet R' \Rightarrow (CInit \Leftrightarrow AInit) \]

2. **Correctness:**
   \[ \forall AState; AState'; CState; CState' \bullet R \land R' \Rightarrow (AOp_i \Leftrightarrow COp_i) \]
Checking refactorings using the Alloy Analyzer

- Using the simplified conditions, we successfully checked refactorings:
  - Inline Method
  - Substitute Algorithm
  - Extract Method
  - Rename
  - Consolidate Conditional Expression
Results

- Translation from Z into Alloy is mostly straightforward
  - Typical problems: integers, infinite data types, schema operators

- Use of existential quantifier is problematic
  - Found *workaround* to this problem when checking refactorings

- Open questions:
  - Does assumption of a total bijective representation relation prohibits the checking of practically relevant refactorings?
  - Compare performance of Alloy Analyzer with other verification tools.
Thank you for your attention!