Generating JML Specifications from Alloy Expressions

Daniel Grunwald[1] Christoph Gladisch[1] Tianhai Liu[1]
Mana Taghdiri[1] Shmuel Tyszberowicz[2]

[1]Karlsruhe Institute of Technology, Germany
[2]Academic College Yafo, Israel

Haifa Verification Conference HVC 2014
Haifa, November 19th, 2014

Motivation

Goals:

- Verification and testing, mainly of linked data structures
- Using JML (the specification language of many tools)

Problems:

- JML specifications of linked data structures are complicated
- Hard to write specifications that are correct and usable for verification and testing

Approach:

- Generate JML specification that are readable and suitable for verification from a simpler notation: Alloy expressions

Motivating example – Specification of `add(Entry d)`

```
class Entry { Entry next; Data data; }
```

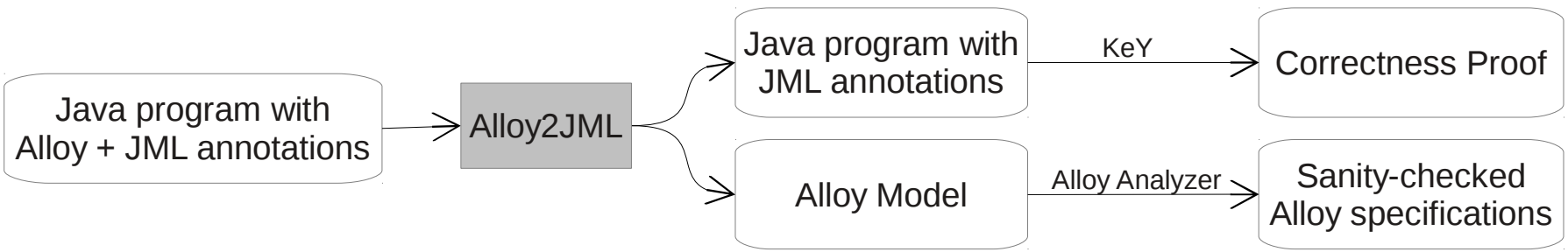
JML Specification (Java Modeling Language):

```
(\forall Data x;
   (\exists Entry a, int i; i>0 &&
    hasNext(this.head,i,a) && a.data==x)
 <==>
   (x==d || (\exists Entry b, int j;
           j>0 && \old(hasNext(this.head,j,b)) && \old(b.data==x)))
```

Alloy Specification:

```
this.head'.^next'.data' = this.head.^next.data + d;
```

Alloy2JML


Java program with
Alloy + JML annotations
Alloy2JML
Java program with
JML annotations
KeY
Correctness Proof
Alloy Model
Alloy Analyzer
Sanity-checked
Alloy specifications

Alloy: A Relational Specification Language

```
class Entry {
 Entry next;
 Data data;
}
```

$\approx$

$$\texttt{Entry}, \texttt{Data} \subseteq \texttt{Object}$$
$$\texttt{next} \subseteq \texttt{Entry} \times \texttt{Entry}$$
$$\texttt{data} \subseteq \texttt{Entry} \times \texttt{Data}$$

Example:

this	head	next	data
$(o1)$	$(o1, o2)$	$(o2, o7)$	$(o2, o10)$
	$(o3, o4)$	$(o7, o8)$	$(o7, o11)$
	$(o5, null)$	$(o8, null)$	$(o8, o13)$

Alloy: A Relational Specification Language

```
class Entry {
 Entry next;
 Data data;
}
```

$\approx$

$$\texttt{Entry}, \texttt{Data} \subseteq \texttt{Object}$$
$$\texttt{next} \subseteq \texttt{Entry} \times \texttt{Entry}$$
$$\texttt{data} \subseteq \texttt{Entry} \times \texttt{Data}$$

Example:

this	head	next	data
$(o1)$	$(o1, o2)$	$(o2, o7)$	$(o2, o10)$
	$(o3, o4)$	$(o7, o8)$	$(o7, o11)$
			$(o8, o13)$

Alloy: A Relational Specification Language

```
class Entry {
 Entry next;
 Data data;
}
```

$\approx$

$$\texttt{Entry}, \texttt{Data} \subseteq \texttt{Object}$$
$$\texttt{next} \subseteq \texttt{Entry} \times \texttt{Entry}$$
$$\texttt{data} \subseteq \texttt{Entry} \times \texttt{Data}$$

Example:

this	head	next	data
$(o1)$	$(o1, o2)$	$(o2, o7)$	$(o2, o10)$
	$(o3, o4)$	$(o7, o8)$	$(o7, o11)$
			$(o8, o13)$

$$\texttt{this.head} = \{(o2)\}$$
$$\texttt{\^{}next} = \{(o2, o7), (o7, o8), (o2, o8)\}$$
$$\texttt{this.head.\^{}next} = \{(o2, o7), (o2, o8)\}$$
$$\texttt{this.head.\^{}next.data} = \{(o11), (o13)\}$$

The Generated JML Specification

```
class Entry { Entry next; Data data; }

/*@ ensures
 (\forall Data x;
    (\exists Entry a, int i; i>0 &&
     hasNext(this.head,i,a) && a.data==x)
  <==>
    (x==d || (\exists Entry b, int j;
            j>0 && \old(hasNext(this.head,j,b)) && \old(b.data==x))))
@*/
public add(Entry d){...};
```

- Specification approach using queries [Gladisch, Tyszberowicz '13]
- Use basic JML subset ($\rightarrow$ compatibility with tools)
- Automatic induction proofs over recursive step of queries
- Compatible with verification and testing

Translation Function – First Attempt

Approach:

- Expand relational operators to their semantic definitions
- Translation function $\mathcal{E}(\texttt{r} \| t_1, ..., t_n)$, true iff $\langle t_1, ..., t_n \rangle \in \texttt{r}$

Example:

$$\mathcal{E}(\texttt{u+v} \| obj) := \mathcal{E}(\texttt{u} \| obj) \,||\, \mathcal{E}(\texttt{v} \| obj)$$

Relational expressions must be in the context of a formula to be fully resolved

Translation Function – First Attempt

Example of translation rules:

$$
\begin{array}{lcl}
\mathcal{B}(\texttt{no r}) & := & \texttt{!(\exists Object x; } \mathcal{E}(\texttt{r} \| \texttt{x})\texttt{)} \\
\mathcal{E}(\texttt{v} \| \textit{val}) & := & \texttt{v == } \textit{val} \\
\mathcal{E}(\texttt{r + s} \| \textit{objs}) & := & \mathcal{E}(\texttt{r} \| \textit{objs}) \texttt{ || } \mathcal{E}(\texttt{s} \| \textit{objs})
\end{array}
$$

Example of a transformation:

$$
\begin{array}{lcl}
\mathcal{B}(\texttt{no u+v}) & = & \texttt{!(\exists Object x;} \mathcal{E}(\texttt{u+v} \| \texttt{x})\texttt{)} \\
& = & \texttt{!(\exists Object x;} \mathcal{E}(\texttt{u} \| x) \texttt{||} \mathcal{E}(\texttt{v} \| x)\texttt{)} \\
& = & \texttt{!(\exists Object x; u==x || v==x)}
\end{array}
$$

Translation Function – First Attempt

Translation rule for transitive closure:

$\mathcal{E}(\texttt{\^{}r} \| obj_1, obj_2) := \texttt{(\exists int i; 0<i; hasR(}obj_1\texttt{,i,}obj_2\texttt{))}$

Example of `hasR` (simplified):

```
/*@ public normal_behavior
 ensures depth==0 ==>(\result==(root==node));
 ensures depth>0 ==>(\result==(hasNext(root.next, depth-1, entry));
@*/
static boolean hasNext(Entry root, int depth, Entry node){...};
```

Problem: $\mathcal{E}$ is not expressive enough to generate contract of `hasR`

Generalized Translation Function – Final Approach

Approach: Generalization of translation function $\mathcal{E}$ with predicates

$\mathcal{E}(\mathtt{r} \| p_1, .., p_n)$ is true iff $\exists(\mathtt{t}_1, .., \mathtt{t}_n) \in \mathtt{r}$: $p_1(c(\mathtt{t}_1)) \wedge .. \wedge p_n(c(\mathtt{t}_n))$

Predicates we used (simplified):

- $\mathit{lift}(\mathtt{obj})(e) := e\mathtt{==obj}$
- $\mathit{nonnull}(e) := e\mathtt{!=null}$
- $\mathit{headrec}_{\mathtt{r}} := \mathtt{(}e\ \mathtt{instof}\ T\ \mathtt{\&\&}\ \mathtt{hasR(}e\mathtt{,steps-1,node))}$
- $\mathit{tailrec}_{\mathtt{r}} := \mathtt{(}e\ \mathtt{instof}\ T\ \mathtt{\&\&}\ \mathtt{hasR(root,steps-1,}e\mathtt{))}$

Example of the Translation Function with Predicates

Example: For arity 1, the *lift*(`obj`) predicate with the $\mathcal{E}$ function corresponds to membership semantics:

$$\begin{aligned} \mathcal{E}(\texttt{r} \| \mathit{lift}(\texttt{obj})) &\iff \exists e \in \texttt{r}:\ \mathit{lift}(\texttt{obj})(e) \\ &\iff \exists e \in \texttt{r}:\ (e == \texttt{obj}) \\ &\iff \texttt{obj} \in \texttt{r} \end{aligned}$$

I.e., this approach can emulate the "First Approach"

Transitive Closure

Translation rule:

$\mathcal{E}(\texttt{\^{}r} \| p_1, p_2) :=$

```
(\exists T_1[^r] obj1, obj2; p1(obj1) && p2(obj2) &&
(\exists int steps; steps>0; hasR(obj1, steps, obj2)))
```

Definition of `hasR` (simplified):

```
/*@ public normal_behavior
ensures steps<0 ==>\result==false;
ensures steps==0==>(\result<==>root==node );
ensures steps>0 ==>(\result<==>E(r||lift(root), headrec_r));
ensures steps>0 ==>(\result<==>E(r||tailrec_r, lift(node)));
measured_by steps;
static pure boolean hasR(T root, int steps, T node); */
```

The Semantics of `null`

`null` marks a field as an empty set

Example 1:

$$\mathcal{B}(\texttt{no this.next}) = \texttt{this.next == null}$$

Example 2:

$$\mathcal{B}(\texttt{no left \& right}) \neq \texttt{(\forall Tree obj; obj.left==null \&\& obj.right==null)}$$

The Semantics of `null`

`null` marks a field as an empty set

Example 1:

$$\mathcal{B}(\texttt{no this.next}) = \texttt{this.next == null}$$

Example 2:

$$\mathcal{B}(\texttt{no left \& right})$$
$$=$$
$$\texttt{(\forall Tree obj; obj.left!=null ==> obj.right!=obj.left)}$$

Evaluation

- Verified methods of LinkedList and BinarySearchTree (BST) classes
- First deductive source code verification of BST with standard JML
- Generated specification is also compatible with other JML tools

Example:

(a) Alloy

```
/*$ensures this.*(left' + right').value' = (this.*(left + right).value) + v;
  ...*/
```

(b) JML Translation

```
/*@ensures (\forall int o1; (\exists Tree o2,int i;
   i>=0 && hasLR(this,i,o2) && o2.value==o1)
   <==>
   \old((\exists Tree o3,int j; j>=0 && hasLR(this,j,o3) && o3.value==o1))||v==o1);
```

Conclusion

Alloy vs. JML

- Alloy simplifies specification of linked data structures
- JML is appropriate for Java-specific details, arithmethics, etc.
- JML is supported by verification and testing tools

Results of this Research

- One elegant translation function suitable for both: simple expressions and transitive closure
- Proved correctness of most rules using Isabelle/HOL
- Generated JML specs are readable and understandable
- First verification of BST methods with standard JML