Generating JML Specifications from Alloy Expressions[†]

Daniel Grunwald, Christoph Gladisch, Tianhai Liu, Mana Taghdiri, and Shmuel Tyszberowicz

Karlsruhe Institute of Technology, Germany
{christoph.gladisch,tianhai.liu,mana.taghdiri}@kit.edu,tyshbe@tau.ac.il

Abstract. Java Modeling Language (JML) is a specification language for Java programs, that follows the design by contract paradigm. However, it is not always easy to use JML, for example when specifying properties of linked data structures. Alloy, on the other hand, is a relational specification language with a built-in transitive closure operator, which makes it particularly suitable for writing concise specifications of linked data structures. This paper presents Alloy2JML, a tool that generates JML specifications from Alloy expression, in order to support both Alloy and JML specifications in the KeY verification engine. This translation allows Java programs with Alloy specifications to be fully verified for correctness. Moreover, Alloy2JML lets Alloy specifications be employed in a variety of tools that accept only JML as their specification language. Supporting Alloy has the additional advantage that users can validate the specifications beforehand using the Alloy Analyzer.

Keywords: JML, Alloy, Java, Theorem proving, KeY, Relational logic

1 Introduction

The ability to write concise and readable specifications highly affects the efficiency of program verification. Providing correct formal specifications can be as difficult as implementing the code correctly. A suitable formalism for specifying program properties not only makes the task of providing specifications easier, but also reduces the likelihood of making mistakes. However, no single specification language is optimal for specifying all possible properties.

JML [21] is a behavioral interface specification language for Java, that adds first-order logic constructs to Java expressions. JML integrates seamlessly into Java and is supported by a wide range of tools for specification type-checking, runtime debugging, static analysis, and verification [4]. JML provides a rich set of specification facilities, yet JML specifications tend to be close to the implementation. Specifying and verifying operations on linked data structures are difficult in JML. Such operations have been specified in JML, e.g. in [1, 24], but no deductive verification of them has been reported. To enable verification, extensions of JML have been used [3].

[†] This work has been partially supported by GIF (grant No. 1131-9.6/2011)

To our knowledge, in [15] we provided the first specification of list operations in *standard* JML that was deductively verified. The approach is to use recursively defined queries (also called observer methods) in specifications to express reachability in linked lists. Consider, for example, a method `add` that inserts the data `d` into a singly linked list starting from the entry `this.head`. Using the approach in [15] we would write the following JML formula to specify that every data `x` in the resulting list is either `d` or was already in the original list:

```
(\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)))) (1)
```

The query `hasNext` expresses that an object `a` can be reached from the object `head` in `i` steps (by traversing the field `next`). This specification approach has the following advantages: (a) it uses a basic subset of JML which enables compatibility with various tools (e.g., [6,8,22]); (b) it allows automatic construction of proofs by induction over the integer provided to the query as the second argument (here `i` and `j`); (c) it does not require ghost-states or ghost-fields, thus makes the specification easier to understand; and (d) the specifications can be used both for deductive verification and for testing[1]. The query is similar to the JML `reach` clause, yet it provides more flexibility; some tools do not support the `reach` clause (e.g. [23, 26]), and those that do support it interpret it differently (e.g., [8, 22]). However, writing such JML specifications is error-prone as they contain technical details. For set-based specifications, it would be easier to use a notation that hides those details and focuses only on the abstract properties.

Alloy [18] is a lightweight declarative specification language for expressing structure-rich software systems. It is based on first-order relational logic, and has built-in operators for transitive closure, set cardinality, integer arithmetic, and set comprehension. Several tools (e.g., [7, 25]) support Alloy as a specification language for Java programs. The transitive closure operator enables users to write concise specifications of linked data structures. The relational override operator allows compact specification of frame conditions. Furthermore, when appropriate, relational specifications let users easily abstract away from the exact order and connection of elements in a data structure by viewing it as a set. The above example can be concisely expressed in Alloy as follows:

```
this.head'.^next'.data' = this.head.^next.data + d;        (2)
```

where ^next denotes the transitive closure over the field `next` (i.e., all nodes reachable by traversing `next`), `+` denotes set union, and unprimed and primed symbols refer to the pre- and post-state of the method, respectively. With this notation we provide a succinct and intuitive representation of set-based specifications. This notation is shorter, easier to understand, and less error-prone than its JML counterpart.

[1] When used for testing the quantifiers have to be bound.

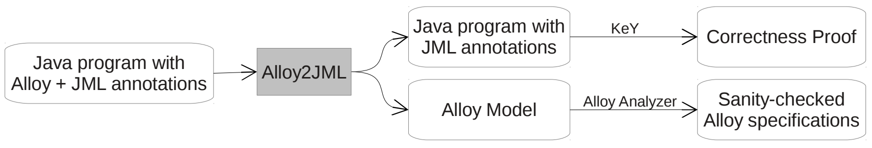
Java program with Alloy + JML annotations
Alloy2JML
Java program with JML annotations
KeY
Correctness Proof
Alloy Model
Alloy Analyzer
Sanity-checked Alloy specifications

Fig. 1. Alloy2JML

JML and Alloy offer complementary views. JML allows to provide detailed Java-specific annotations and to utilize Java for that. It is also suitable for specifying arithmetic properties, and certain properties of data structures where the position of elements is important. Other properties, e.g. those that constrain the set of all the elements of a linked data structure, are easier to express in Alloy.

This paper describes Alloy2JML, a tool that translates Alloy specifications of Java programs into JML. The automatic translation is particularly beneficial for properties that are difficult or error-prone to express in JML directly. It takes programs in which each property is specified in either JML or Alloy, and translates Alloy specifications to JML to yield uniform JML specifications. The translation lets Java programs with Alloy specifications be fully verified for correctness using the KeY verification engine [2]. Alloy2JML's output conforms to the format suggested in [15], thus offers all the advantages listed above. For example, it generates the JML specification (1) from the Alloy formula (2). The output is essentially in standard JML—minor differences exist to support KeY, which can be eliminated by simple syntactic changes. We have proved the correctness of our translation [16] for a subset of Alloy using Isabelle/HOL [20].

We aim at producing JML formulas that are both usable for verification and human-readable. Readability is particularly important when using interactive verification tools such as KeY. It is not only necessary for debugging failed proof attempts, but also for providing additional lemmas in the proof process. To generate readable specifications, we use a translation function that tries to minimize the syntactic scope of quantifiers by delaying the introduction of quantification guards. A subsequent simplification step eliminates most of the redundant quantifiers.

In [15] we showed only specification examples and did not provide a *systematic* way of defining the queries and how to write the specifications. Its applicability to trees was also an open question. Here, we generalize and extend that work. The translation provides a systematic way of deriving JML specifications for arbitrary linked data structures from the more abstract Alloy specifications and it eliminates the error-prone task of manually defining of queries. The queries can be applied not only over a single field but also over an arbitrary relation denoted by an Alloy expression.

As shown in Fig. 1, the input to Alloy2JML is a Java program annotated with both JML and Alloy specifications, and the output is a Java program annotated with JML specifications only. Alloy2JML also outputs an Alloy model that declares Alloy signatures for the classes in the input program and Alloy predicates for methods' specifications (see [16]). Using this model, the Alloy Analyzer [18]—an automatic tool that checks Alloy models within bounded domains—can be

```
exp ::= id | id' | freshType                 form ::= exp in exp | exp = exp
| none | exp + exp| exp & exp                | exp  (> | >= | < | <=) exp
| exp - exp | exp . exp                      | form (and | or | implies | iff) form
| exp -> exp | exp ++ exp | ~exp             | not form | (no | some | lone) exp
| ^exp | {[id: expr]+ | form}                | (all | some) [id : exp]+ | form
| number | #exp | (sum id: exp | exp)
```

Fig. 2. Abstract syntax for translated Alloy expressions (exp) and formulas (form)

used to sanity-check the Alloy specifications prior to performing the full (possibly interactive) verification. For example, it can help detect under-specification and errors by visualizing instances that satisfy the specifications, and detect over-specification by showing the unsatisfiable core. This makes Alloy a particularly attractive specification language compared to other languages that support sets and relations.

2 Background

2.1 Alloy

Alloy is a specification language based on first-order logic [18]. Every Alloy expression evaluates to a relation. Unary relations are declared as signatures, and represent sets of atoms. Relations with higher arities are declared as fields and represent sets of tuples. The constant `none` denotes the empty set. Set operations union, intersection, and difference are denoted by `+`, `&`, and `-`, respectively. For relations `r` and `s`, the relational join and Cartesian product are denoted by `r.s` and `r->s`, respectively. The relational override `r++s` contains all tuples of `s`, and those tuples of `r` whose first element does not appear as the first element of any tuple in `s`. The expression `~r` denotes the transpose of `r`, and the transitive closure `^r` defines the smallest transitive relation that contains `r`. Set comprehensions make relations with all tuples for which a certain formula holds. The Alloy integer type, `Int`, represents the set of integer atoms. All integers (including numbers, the result of the set cardinality operator `#`, and the `sum` quantifier) are treated as sets of integer atoms (Alloy 4.2). Arithmetic operators are defined as functions in the Alloy integer library (`add[a,b]` and `sub[a,b]`). The expression `(sum x: S | e)` computes the sum of the values that the integer expression `e` can take for all distinct bindings of the variable `x` in `S`.

Basic formulas are constructed using the operators `in` (subset), `=` (equality), and integer comparators. They are combined using the standard logical operators. The multiplicity formulas `no r`, `some r`, and `lone r` constrain `r` to have zero, at least one, and at most one tuple, respectively. The quantifiers `all` and `some` denote the universal and existential quantifiers. It should be noted that the Alloy Analyzer supports higher-order quantification when the quantifier can be eliminated through skolemization. We, however, do not support higher-order quantifications at all as they cannot be translated to JML.

We let Java programs be annotated with legal Alloy formulas. We provide special translation rules for the Alloy constructs of Fig. 2, and desugar all others to this subset. Figure 2 slightly deviates from Alloy by introducing extra identifiers that have special meanings in our translation: a primed identifier refers to the post-state of a method, whereas an unprimed one refers to its pre-state. `freshT` denotes the set of objects of type `T` that are allocated in the post-state, but not in the pre-state[2].

2.2 JML

Java Modeling Language (JML) [21] is a first-order, behavioral interface specification language for Java. Side-effect free Java expressions, standard logical operators, universal and existential quantifiers are allowed in JML annotations. JML also supports various clauses and keywords for better specifications. The ones used by our translation are described below.

The `requires` clause denotes a method's precondition, evaluated in the pre-state of the method call. If a method terminates normally, i.e. without throwing an exception, then the normal post-condition—given in the `ensures` clause—must hold in the post-state. The `normal_behavior` clause specifies that if a method's precondition holds, the method must return normally. The `invariant` clause denotes an object invariant that must hold at the end of each constructor's execution, and at the beginning and end of all non-static methods that are not marked as `helper`. The memory locations (represented by a set of fields) that are listed in the `modifies` clause are the only pre-state locations that can be modified by a method. The `measured_by e` clause is used in a termination argument for a recursive specification, where the integer expression `e` decreases on each iteration and evaluates to zero when the method terminates.

Java expressions used in pre- and post-conditions are evaluated in the heap's pre-state and the post-state, respectively. To access the initial (pre-state) value of an expression `e` in the post-condition, the expression `\old(e)` is used. The keyword `\result` refers to the value returned by a non-void method. The `\fresh(o)` operator constrains the object `o` not to exist in the pre-state and to be non-null in the post-state. Member fields, formal parameters, and return values are considered to be non-null by default. The `nullable` modifier specifies that the `null` value is also acceptable. The modifier `pure` denotes that a method has no side-effects and thus can be used in the annotations. The `model` modifier denotes those fields and methods that can be used only in the annotations.

KeY [2] accepts JML*, a modified version of JML, as the specification language for Java programs. JML* implements most, but not all, JML features and adds a few more. Most relevant to our work is a semantic difference in the interpretation of quantifiers. The range of JML quantifiers extends over all objects of the given type, including those that are not yet created [21]. In JML*, on the other hand, the quantifier ranges over only those objects that have been created in the current heap state. It is possible to obtain the JML* quantifier semantics in JML by introducing predicates that explicitly distinguish between created

[2] Fresh objects could be specified by `T' - T`, but then the translation could not distinguish other set differences from fresh objects (for which it generates `\fresh` clauses).

```
 1 class Entry {
 2  /*@nullable*/ Entry next;
 3  /*@nullable*/ Data data;
 4
 5  //$ensures this.data' = d;
 6  //$ensures no this.next';
 7  //$modifies this.data,this.next;
 8  Entry(/*@nullable*/Data d)
 9  { this.data = d; }
10 }
11 class LinkedList {
12  Entry head;
13  int length;
14
15  //$ensures this.head'.^next'.data'
16  //$ = this.head.^next.data + d;
17  //$ensures this.length'=add[this.length,1];
18  //$modifies this.head.next, this.length;
19  void add(Data d) {
20   Entry newEntry = new Entry(d);
21   newEntry.next = head.next;
22   head.next = newEntry;
23   length++;
24  }
25 }
26 class Data { .. }
```

(a)

```
 1 class Entry {
 2  /*@nullable*/ Entry next;
 3  /*@nullable*/ Data data;
 4
 5  //@ensures this.data == d;
 6  //@ensures this.next == null;
 7  //@modifies this.data,this.next;
 8  Entry(/*@nullable*/Data d){ .. }
 9 }
10 class LinkedList {
11  Entry head;
12  int length;
13
14  //@ensures (\forall Data x;
15  //@ (\exists Entry a, int i;
16  //@   a.data == x && i > 0 &&
17  //@   hasNext(this.head,i,a))
18  //@ <==> (\exists Entry b, int j;
19  //@   \old(b.data == x) && j > 0 &&
20  //@   \old(hasNext(this.head,j,b)))
21  //@   || d == x);
22  //@ensures this.length ==
23              \old(this.length)+1;
24  //@modifies this.head.next, this.length;
25  void add(Data d) { ... }
26 }
27 class Data { .. }
```

(b)

Fig. 3. Example: (a) original, (b) translated

and uncreated objects. Furthermore, the JML* construct `\infinite_union(C o; o.f)`, where `C` is a class and `f` is a field, gives the set of memory locations `o.f` for all objects `o` of class `C`. The construct can be replaced using the model type `JMLDataGroup` in standard JML (cf. [15]) (not included in JML*).

3 Motivating Example

We assume that the Alloy specifications of Java programs are written at the concrete representation level of the code, and follow a *relational view of the heap* [25]. That is, Java types are viewed as Alloy signatures, fields as binary relations, and local variables and parameters as singleton sets.

Figure 3 elaborates the example of Section 1, showing our translation of Alloy to JML. Figure 3(a) gives an implementation of a singly linked list where the `head` and the `length` fields (Lines 12-13) denote the first entry and the number of entries of the list, respectively. The list's first entry is dummy; it does not contain any data, and exists even for an empty list. The `length` field ensures that the list is finite, which is necessary for proving termination of methods that traverse the list. The `add` method (Lines 19-24) inserts the given data `d` at the beginning of the receiver list.

Alloy and JML annotations are marked by `//$` and `//@` respectively. The post-conditions of the `Entry` constructor ensure that the given data `d` is stored in the `data` field of the created entry (Fig. 3(a), Line 5), and that the `next` field of this entry is set to `null` (Line 6). We assume that Alloy specifications model the Java `null` object as an empty set. The first post-condition of the `add` method specifies that the set of data stored in this list in the post-state equals that set in the pre-state, augmented with the added data `d` (Lines 15-16). This example demonstrates that specifications can be arbitrarily partial. This post-condition, for example, does not specify that the given data is inserted at the *beginning* of the list. The second post-condition (Line 17) specifies that `length` is updated properly. The invariants of the `LinkedList` class are omitted for space reasons.

JML specifications produced by Alloy2JML are shown in Fig. 3(b). To handle Alloy's transitive closure operator, we introduce pure Java methods that can be used in JML annotations. For a field `f` of type `T` declared in a class `S`, we define a pure Java method `hasF(C x, int i, C y)` that returns true if `x` is non-null and `y` is reachable from `x` by `i` times following the field `f`, and false otherwise. The type `C` is the first common type of `S` and `T` in the type hierarchy of the analyzed method. In addition to simple relational joins which are translated to field dereferences, the post-condition of `add` (Fig. 3(a), Lines 15-17) contains set equality, set union, and transitive closure operators. Set equality is translated using its definition: any object in the right-hand-side set must be in the left-hand-side one, and vice versa. Set union is handled using disjunction. An expression containing the transitive closure `some o.^next` is translated using `(\exists Entry e, int i; hasNext(o, i, e))`, where the integer `i` can be any positive number. The resulting JML specification is shown in Lines 14-23 of Fig. 3(b).

As shown by this example, we translate Alloy annotations into a basic variant of JML. Alloy annotations are particularly concise and readable when specifications involve reachability and set semantics. More examples can be found in [16].

4 Translation from Alloy to JML

4.1 The Translation Function

We have experimented with several translations and evaluated the applicability and readability of the resulting JML specifications for verification using KeY. In the following, we describe two preliminary approaches (Approach 1, Approach 2) to motivate and explain our solution (Approach 3). For brevity we often use the term *relation* to refer to sets and relations.

Approach 1: Since Alloy expressions evaluate to relations, a direct translation of Alloy to JML requires the notion of relations in JML. Such a translation could be done using a translation function $\mathcal{E}(\texttt{r}) \to e$ that maps an Alloy expression `r` to a JML expression e of a container (or array) type in Java. The translation of a union operation, for instance, would then become $\mathcal{E}(\texttt{r + s}) = \texttt{union(}\mathcal{E}(\texttt{r})\texttt{,}\mathcal{E}(\texttt{s})\texttt{)}$ where `union` is a Java method that operates on containers. KeY expands method invocations to their contracts. Expanding complex expressions, however, leads to very complex verification conditions which we found impractical.

Approach 2: The explicit representation of sets and relations in Java/JML can be omitted by expanding relational operators to their semantic definitions during the translation. For this, we modify the translation function to $\mathcal{E}(\texttt{r} \| t_1, \ldots, t_n) \rightarrow e$ which now maps an Alloy expression `r` along with JML expressions $t_1, ..., t_n$ to a boolean JML expression e, such that e is true iff <$t_1, \ldots, t_n$> corresponds to a tuple in `r`. The double bars $\|$ are a visual aid separating the translated expression (left-hand side) from the expressions that form the tuple (right-hand side). For example, the translation of the union operation can be expressed as $\mathcal{E}(\texttt{u+v} \| obj) := \mathcal{E}(\texttt{u} \| obj)\,\texttt{||}\,\mathcal{E}(\texttt{v} \| obj)$, where, for simplicity, `u` and `v` are unary relations denoting program variables. In isolation, this expression cannot be further resolved to a JML expression as the meta variable *obj* needs to be instantiated. However, it can be resolved in the context of the formula in which it is used. For this, we introduce another translation function, $\mathcal{B}$, which maps Alloy formulas to boolean JML expressions. Consider, for example, the following translation rules:

$\mathcal{B}(\texttt{no r}) := !(\texttt{\\exists Object x; } \mathcal{E}(\texttt{r} \| \texttt{x}))$
$\mathcal{E}(\texttt{v} \| val) := \texttt{v}_c \texttt{ == } val$ *v_c denotes name resolution*
$\mathcal{E}(\texttt{r + s} \| objs) := \mathcal{E}(\texttt{r} \| objs) \texttt{ || } \mathcal{E}(\texttt{s} \| objs)$
$\mathcal{E}(\texttt{\^{}r} \| obj_1, obj_2) := (\texttt{\\exists int i; 0 <= i; hasR(}obj_1\texttt{, i, }obj_2\texttt{))}$

Using the first three rules, the expression `no u+v` will be translated to the JML expression `!(\exists Object x; u==x || v==x)` without explicitly using Java containers. The last rule shows our basic idea for translating transitive closure. We express the reachability of obj_2 from obj_1 via the relational expression `r`, i.e. $(obj_1, obj_2) \in \hat{}r$, using a boolean query method `hasR`. The integer `i` stands for the number of times that `r` is traversed in order to reach obj_2 when starting from obj_1. This is used as the induction variable in induction proofs. However, to generate `hasR` from `r`, the translation function $\mathcal{E}$ has to be generalized further.

Approach 3: In [15] we have described how a user can manually write a query (such as `hasR`) for a list data structure. Here we describe a general method to automatically translate the transitive closure of an arbitrary expression `r` into a recursive definition of the query. Rather than introducing another translation function for this purpose, we generalize the function $\mathcal{E}$ to the form $\mathcal{E}(\texttt{r} \| p_1, \ldots, p_n)_c$ where `r` is the Alloy relational expression to be translated; $p_1, \ldots, p_n$ is a list of translation predicates applicable to JML expressions; and c is a translation context capturing various information. The context provides a mapping from Alloy types, relations, and variables to their corresponding symbols in JML, and tells whether the expression is evaluated in the pre- or post-state (the latter generates expressions embedded in `\old(...)`). The number of predicates (n) must match the arity of the relation `r`, and the predicates must be well-formed. The semantics of the translation function is defined by:
$\mathcal{E}(\texttt{r} \| p_1, .., p_n)_c$ evaluates to `true` $\iff \exists(\texttt{t}_1, .., \texttt{t}_n) \in \texttt{r}\texttt{:}\; p_1(c(\texttt{t}_1)) \wedge .. \wedge p_n(c(\texttt{t}_n))$

The predicates are a generalization of the terms $t_1, ..., t_n$ described in Approach 2. While a term can represent only one element, a predicate can represent a set of

$\mathbf{R}_1$:	$\mathcal{E}(\texttt{v}\Vert p_1)_c$	$:= p_1(c(\texttt{v}))$
$\mathbf{R}_2$:	$\mathcal{E}(\texttt{T}\Vert p_1)_c$	:= (\exists c(T) obj; p_1(obj))
$\mathbf{R}_3$:	$\mathcal{E}(\texttt{member}\Vert p_1, p_2)_c$	:= (\exists $\mathcal{T}_1$[member] obj; p_1(obj) && p_2(obj.c(member)))
$\mathbf{R}_4$:	$\mathcal{E}(\texttt{none}\Vert p_1)_c$	:= false
$\mathbf{R}_5$:	$\mathcal{E}(\texttt{n}\Vert p_1)_c$	$:= p_1(\texttt{n})$
$\mathbf{R}_6$:	$\mathcal{E}(\texttt{r.s}\Vert p_1, ..., p_{n+m})_c$	:= (\exists $\mathcal{T}_1$[s] obj; $\mathcal{E}(\texttt{r}\Vert p_1, ..., p_n, \mathit{lift}(\texttt{obj}))_c$ && $\mathcal{E}(\texttt{s}\Vert \mathit{lift}(\texttt{obj}), p_{n+1}, ..., p_{n+m})_c$) where $n = \mathit{arity}(\texttt{r}) - 1$ and $m = \mathit{arity}(\texttt{s}) - 1$
$\mathbf{R}_7$:	$\mathcal{E}(\texttt{r + s}\Vert p_1, ..., p_n)_c$	:= ($\mathcal{E}(\texttt{r}\Vert p_1, ..., p_n)_c$ || $\mathcal{E}(\texttt{s}\Vert p_1, ..., p_n)_c$)
$\mathbf{R}_8$:	$\mathcal{E}(\texttt{r \& s}\Vert p_1, ..., p_n)_c$	:= (\exists $\mathcal{T}_1$[r&s] o$_1$,..., $\mathcal{T}_n$[r&s] o$_n$;p_1(o$_1$) &&...&& p_n(o$_n$) && && $\mathcal{E}(\texttt{r}\Vert \mathit{lift}(\texttt{o}_1), ..., \mathit{lift}(\texttt{o}_n))_c$ && $\mathcal{E}(\texttt{s}\Vert \mathit{lift}(\texttt{o}_1), ..., \mathit{lift}(\texttt{o}_n))_c$)
$\mathbf{R}_9$:	$\mathcal{E}(\texttt{r - s}\Vert p_1, ..., p_n)_c$	:= (\exists $\mathcal{T}_1$[r-s] o$_1$,..., $\mathcal{T}_n$[r-s] o$_n$;p_1(o$_1$) &&...&& p_n(o$_n$) && $\mathcal{E}(\texttt{r}\Vert \mathit{lift}(\texttt{o}_1), ..., \mathit{lift}(\texttt{o}_n))_c$ && !$\mathcal{E}(\texttt{s}\Vert \mathit{lift}(\texttt{o}_1), ..., \mathit{lift}(\texttt{o}_n))_c$)
$\mathbf{R}_{10}$:	$\mathcal{E}(\texttt{r ++ s}\Vert p_1, ..., p_n)_c$	:= (\exists $\mathcal{T}_1$[r++s] obj; p_1(obj) && ($\mathcal{E}(\texttt{b}\Vert \mathit{lift}(\texttt{obj}), p_2, ..., p_n)_c$|| ($\mathcal{E}(\texttt{r}\Vert \mathit{lift}(\texttt{obj}), p_2, ..., p_n)_c$ && !$\mathcal{E}(\texttt{s}\Vert \mathit{lift}(\texttt{obj}), \underbrace{\mathit{nonnull}, ...}_{n-1 \text{ times}})_c$)))
$\mathbf{R}_{11}$:	$\mathcal{E}(\texttt{r -> s}\Vert p_1, ..., p_{n+m})_c$	:= ($\mathcal{E}(\texttt{r}\Vert p_1, ..., p_n)_c$ && $\mathcal{E}(\texttt{s}\Vert p_{n+1}, ..., p_{n+m})_c$) where $n = \mathit{arity}(\texttt{r})$ and $m = \mathit{arity}(\texttt{s})$
$\mathbf{R}_{12}$:	$\mathcal{E}(\texttt{\~{}r}\Vert p_1, ..., p_n)_c$	$:= \mathcal{E}(\texttt{r}\Vert p_n, ..., p_1)_c$
$\mathbf{R}_{13}$:	$\mathcal{E}(\{\texttt{v}_1 : \texttt{r}_1, ..., \texttt{v}_n : \texttt{r}_n \mid \texttt{F}\}\Vert p_1, ..., p_n)_c$	:= (\exists $\mathcal{T}_1$[v$_1$] o$_1$, ..., $\mathcal{T}_1$[v$_n$] o$_n$; p_1(o$_1$) && ... && p_n(o$_n$) && $\mathcal{E}(\texttt{r}_1\Vert \mathit{lift}(\texttt{o}_1))_{c_1}$ && ... && $\mathcal{E}(\texttt{r}_n\Vert \mathit{lift}(\texttt{o}_n))_{c_n}$ && $\mathcal{B}(\texttt{F})_{c_{n+1}}$)
$\mathbf{R}_{14}$:	$\mathcal{E}(\texttt{\#r}\Vert p_1)_c$	:= p_1(\num_of $\mathcal{T}_1$[r] o$_1$, ..., $\mathcal{T}_n$[r] o$_n$; $\mathcal{E}(\texttt{r}\Vert \mathit{lift}(\texttt{o}_1), ..., \mathit{lift}(\texttt{o}_n))_c$)
$\mathbf{R}_{15}$:	$\mathcal{E}(\texttt{sum v: r} \mid \texttt{i}\Vert p_1)_c$	:= p_1(\sum $\mathcal{T}_1$[r] obj; $\mathcal{E}(\texttt{r}\Vert \mathit{lift}(\texttt{obj}))_c$; $\mathcal{I}(\texttt{i})_{c'}$)

$\mathbf{R}_{16}$:	$\mathcal{B}(\texttt{F and G})_c$	:= ($\mathcal{B}(\texttt{F})_c$ && $\mathcal{B}(\texttt{G})_c$)
$\mathbf{R}_{17}$:	$\mathcal{B}(\texttt{F or G})_c$	:= ($\mathcal{B}(\texttt{F})_c$ || $\mathcal{B}(\texttt{G})_c$)
$\mathbf{R}_{18}$:	$\mathcal{B}(\texttt{!F})_c$	:= (!$\mathcal{B}(\texttt{F})_c$)
$\mathbf{R}_{19}$:	$\mathcal{B}(\texttt{F iff G})_c$	:= ($\mathcal{B}(\texttt{F})_c$ <==> $\mathcal{B}(\texttt{G})_c$)
$\mathbf{R}_{20}$:	$\mathcal{B}(\texttt{F implies G})_c$	:= ($\mathcal{B}(\texttt{F})_c$ ==> $\mathcal{B}(\texttt{G})_c$)
$\mathbf{R}_{21}$:	$\mathcal{B}(\texttt{i } \mathit{op} \texttt{ j})_c$	:= ($\mathcal{I}(\texttt{i})_c$ op $\mathcal{I}(\texttt{j})_c$) where $\mathit{op} \in \{<, >, <=, >=\}$
$\mathbf{R}_{22}$:	$\mathcal{B}(\texttt{r in s})_c$	:= (\forall $\mathcal{T}_1$[r+s] o$_1$, ..., $\mathcal{T}_n$[r+s] o$_n$; $\mathcal{E}(\texttt{r}\Vert \mathit{lift}(\texttt{o}_1), ..., \mathit{lift}(\texttt{o}_n))_c$ ==> $\mathcal{E}(\texttt{s}\Vert \mathit{lift}(\texttt{o}_1), ..., \mathit{lift}(\texttt{o}_n))_c$)
$\mathbf{R}_{23}$:	$\mathcal{B}(\texttt{r = s})_c$	:= (\forall $\mathcal{T}_1$[r+s] o$_1$, ..., $\mathcal{T}_n$[r+s] o$_n$; $\mathcal{E}(\texttt{r}\Vert \mathit{lift}(\texttt{o}_1), ..., \mathit{lift}(\texttt{o}_n))_c$ <==> $\mathcal{E}(\texttt{s}\Vert \mathit{lift}(\texttt{o}_1), ..., \mathit{lift}(\texttt{o}_n))_c$)
$\mathbf{R}_{24}$:	$\mathcal{B}(\texttt{no r})_c$	:= (!$\mathcal{E}(\texttt{r}\Vert \underbrace{\mathit{nonnull}}_{\mathit{arity}(\texttt{r}) \text{ times}})_c$)
$\mathbf{R}_{25}$:	$\mathcal{B}(\texttt{some r})_c$	$:= \mathcal{E}(\texttt{r}\Vert \underbrace{\mathit{nonnull}}_{\mathit{arity}(\texttt{r}) \text{ times}})_c$
$\mathbf{R}_{26}$:	$\mathcal{B}(\texttt{lone r})_c$	:= (\forall $\mathcal{T}_1$[r] o$_1$, ..., $\mathcal{T}_n$[r] o$_n$, $\mathcal{T}_1$[r] w$_1$, ..., $\mathcal{T}_n$[r] w$_n$; ($\mathcal{E}(\texttt{r}\Vert \mathit{lift}(\texttt{o}_1), ..., \mathit{lift}(\texttt{o}_n))_c$ && $\mathcal{E}(\texttt{r}\Vert \mathit{lift}(\texttt{w}_1), ..., \mathit{lift}(\texttt{w}_n))_c$) ==> (o$_1$ == w$_1$ && ... && o$_n$ == w$_n$))
$\mathbf{R}_{27}$:	$\mathcal{B}(\texttt{all v : r} \mid \texttt{F})_c$	:= (\forall $\mathcal{T}_1$[r] obj; $\mathcal{E}(\texttt{r}\Vert \mathit{lift}(\texttt{obj}))_c$ ==> $\mathcal{B}(\texttt{F})_{c*}$)
$\mathbf{R}_{28}$:	$\mathcal{B}(\texttt{some v : r} \mid \texttt{F})_c$	:= (\exists $\mathcal{T}_1$[r] obj; $\mathcal{E}(\texttt{r}\Vert \mathit{lift}(\texttt{obj}))_c$ && $\mathcal{B}(\texttt{F})_{c*}$)

$\mathbf{R}_{29}$:	(\exists T obj; f_1 && x==obj && f_2(obj))	$\hookrightarrow$ (f_1 && x instanceof T && f_2((T)x))
$\mathbf{R}_{30}$:	(\forall T obj; f_1 && x==obj ==> f_2(obj))	$\hookrightarrow$ (f_1 && x instanceof T ==> f_2((T)x))
$\mathbf{R}_{31}$:	(\forall T obj; f_1 && x==obj <==> f_2 && y==obj)	$\hookrightarrow$ ((f_1 ? x : null) == (f_2 ? y : null))
$\mathbf{R}_{32}$:	(x instanceof T)	$\hookrightarrow$ (x != null) if the type of x is a subtype of T
$\mathbf{R}_{33}$:	((T)x)	$\hookrightarrow$ (x) if the type of x is equal to T
$\mathbf{R}_{34}$:	(x != null)	$\hookrightarrow$ true if x is statically known to be non-null

Table 1. The translation functions $\mathcal{E}$ and $\mathcal{B}$, and simplification rules R_{29} – R_{34}. `v` is an Alloy variable, `T` is a type signature, `member` is an Alloy relation for a Java field, `n` is an integer literal, `r, s` are relational Alloy expressions, `i, j` are integer Alloy expressions, `F, G` are Alloy formulas, $\mathcal{T}_i$ gives the JML type corresponding to the type of the i^{th} column of the given relation, the translation contexts $c_i, c', c*$ are extensions of c with the mappings from the Alloy variables to the JML variables.

elements. This generalization allows a concise and unified translation of simple expressions as well as expressions with transitive closure. It also improves the readability of the resulting JML expressions, because the predicates are used as quantification guards and are propagated to subexpressions where they are needed, thus quantifiers can be introduced locally near the subexpression. The translation uses the predicates $\mathit{lift}(\texttt{obj})$, $\mathit{nonnull}$, $\mathit{headrec}_{\texttt{r}}$, and $\mathit{tailrec}_{\texttt{r}}$. The latter two are used for transitive closure, and are defined recursively over `r` as described in Section 4.2. The semantics of the former predicates is defined by:

$$\mathrm{R}_{35}\colon \mathit{lift}(\texttt{obj})(e) := e\texttt{==obj} \qquad\qquad \mathrm{R}_{36}\colon \mathit{nonnull}(e) := e\texttt{!=null}$$

For relations of arity 1, using the $\mathit{lift}(\texttt{obj})$ predicate with the $\mathcal{E}$ function corresponds to membership semantics[3]: $\mathcal{E}(\texttt{r}\|\mathit{lift}(\texttt{obj}))_c \iff \exists e \in \texttt{r}\colon\ \mathit{lift}(\texttt{obj})(e) \iff \exists e \in \texttt{r}\colon\ (e\texttt{==obj}) \iff \texttt{obj} \in \texttt{r}$. The usage of the $\mathit{nonnull}$ predicate with the $\mathcal{E}$ function checks whether an Alloy relation is non-empty: $\mathcal{E}(\texttt{r}\|\mathit{nonnull})_c \iff \exists e \in \texttt{r}\colon\ \mathit{nonnull}(e) \iff \exists e \in \texttt{r}\colon\ (e! = \mathit{null}) \iff \texttt{r} \neq \emptyset$. Note that if the value of a dereferenced field, say `o.f`, is non-null, then `o.f` represents a singleton set containing this value. However, if `o.f` is `null`, we treat `o.f` as the empty set (as in [25]), rather than a set containing null as an element (as in [27]). Using `null` as a *marker* for empty sets is convenient since it is the only value that can be assigned to any field of any reference type.

The first two sections of Table 1 define $\mathcal{E}$, for relational expressions, and $\mathcal{B}$, for formulas. An additional third function $\mathcal{I}$ is used for integer expressions. Due to space issues, some rules are omitted. We have proved the correctness of the rules with respect to the semantics of the $\mathcal{E}$ function as given above using Isabelle/HOL [20]. These proofs allowed us to discover and fix subtle problems related to Java heap-states and handling of null references. The complete list of rules, correctness proofs, and further details can be found in [16].

To illustrate the details of the translation rules, consider the translation of the expression "`no this.next`" (using the declarations from Fig. 3).

$$\begin{aligned}
\mathcal{B}(\texttt{no this.next})_c &\overset{\mathrm{R}_{24}}{=} \texttt{(!}\mathcal{E}(\texttt{this.next}\|\mathit{nonnull})_c\texttt{)}\\
&\overset{\mathrm{R}_{6}}{=} \texttt{(!(\textbackslash exists Entry t; }\mathcal{E}(\texttt{this}\|\mathit{lift}(\texttt{t}))_c\texttt{ \&\& }\mathcal{E}(\texttt{next}\|\mathit{lift}(\texttt{t}),\mathit{nonnull})_c\texttt{))}\\
&\overset{\mathrm{R}_{1},\mathrm{R}_{3}}{=} \texttt{(!(\textbackslash exists Entry t; }\mathit{lift}(\texttt{t})(\texttt{this})\\
&\qquad \texttt{\&\& (\textbackslash exists Entry obj; }\mathit{lift}(\texttt{t})(\texttt{obj})\texttt{ \&\& }\mathit{nonnull}(\texttt{obj.next})\texttt{)))}\\
&\overset{\mathrm{R}_{35},\mathrm{R}_{36}}{=} \texttt{(!(\textbackslash exists Entry t; this == t}\\
&\qquad \texttt{\&\& (\textbackslash exists Entry obj; obj == t \&\& obj.next != null)))}
\end{aligned}$$

Both quantifiers are redundant due to the equalities `this==t` and `obj==t`. The translation is followed by a simplification step. The third section of Table 1 shows a subset of our simplification rules (cf. [16] for complete details). This step dramatically increases the readability and analyzability of the resulting JML formulas. Applying the simplifications to the example yields:

$$\begin{aligned}
\mathcal{B}(\texttt{no this.next})_c &\overset{\mathrm{R}_{29}}{=} \texttt{(!(\textbackslash exists Entry t; this == t}\\
&\qquad \texttt{\&\& (t instanceof Entry \&\& ((Entry)t).next != null)))}
\end{aligned}$$

[3] The application of the context c is omitted to improve readability.

$\overset{R_{32},R_{33}}{=}$ `(!(\exists Entry t; this == t`
`&& (t != null && t.next != null)))`

$\overset{R_{34}}{=}$ `(!(\exists Entry t; this == t && t.next != null))`

$\overset{R_{29}}{=}$ `(!(this instanceof Entry && ((Entry)this).next != null))`

$\overset{R_{32},R_{33}}{=}$ `!(this != null && this.next != null)`

$\overset{R_{34}}{=}$ `this.next == null`

Note that although Java null value is assumed to be represented as empty set in Alloy specifications, an empty set in Alloy specifications does not always represent null. E.g., our rules translate the formula “`no left & right`” to:

```
(\forall Tree obj; obj.left != null ==> obj.right != obj.left)
```

where left and right denote the two pointers of a binary tree.

4.2 Transitive Closure

In [15] we have explored how to specify methods of a linked list using a query (i.e., an observer method) `getNext(o,i)` that returns the i*'th* element of the list starting from `o`. Here we generalize that approach to arbitrary data structures, and support the use of complex Alloy expressions with the transitive closure operator. Given an Alloy relational expression `r`, we define the query method `hasR` such that it evaluates to true iff a given object `node` is reachable from object `root` via the relation `r` in `steps` number of steps:[4]

```
/*@ public normal_behavior
ensures steps < 0 ==> \result == false;
ensures steps == 0 ==> (\result <==> root==node && root != null);
ensures steps > 0 ==> (\result <==> root!=null && $\mathcal{E}(\texttt{r}\|\mathit{lift}(\texttt{root}), \mathit{headrec}_{\texttt{r}})_c$);
ensures steps > 0 ==> (\result <==> node!=null && $\mathcal{E}(\texttt{r}\|\mathit{tailrec}_{\texttt{r}}, \mathit{lift}(\texttt{node}))_c$);
measured_by steps;
static model helper pure
boolean hasR(nullable $T$ root, int steps, nullable $T$ node); */
```

In this definition, $T \times T$ is the JML type corresponding to the type of `^r` as determined by Alloy's type inference. The translation predicates $\mathit{headrec}_{\texttt{r}}$ and $\mathit{tailrec}_{\texttt{r}}$ recursively call the method `hasR` and are defined as:

$$\mathit{headrec}_{\texttt{r}}(e) := \texttt{(}e\ \texttt{instanceof}\ T\ \texttt{\&\& hasR((}T\texttt{)}e\texttt{, steps-1, node))}$$
$$\mathit{tailrec}_{\texttt{r}}(e) := \texttt{(}e\ \texttt{instanceof}\ T\ \texttt{\&\& hasR(root, steps-1, (}T\texttt{)}e\texttt{))}$$

In the head-recursive `ensures` clause, the $\mathcal{E}$ function is used to produce a JML expression that evaluates to `true` if the relation `r` contains a pair $(\texttt{root}, e)$, i.e. e is reachable from `root` in one step, and `node` is reachable from e in `steps-1` steps.

[4] `R` is always a unique name for `r`, e.g. `left + right` gets the name `LeftUnionRight`.

Similarly, the tail-recursive postcondition uses $\mathcal{E}$ to produce a JML expression that evaluates to `true` if the relation `r` contains a pair $(e,$ `node`$)$ and e is reachable from `root` in `steps-1` steps. For specification, it is sufficient to use either head- or tail-recursion, but having both sometimes simplifies the verification. Using the `hasR` query definition, we can translate the transitive closure as follows:

R$_{37}$: $\mathcal{E}(\texttt{\^{}r} \| p_1, p_2)_c :=$ `(\exists` $\mathcal{T}_1$`[^r] obj1, obj2;` p_1`(obj1) &&` p_2`(obj2) &&`
`(\exists int steps; steps>0; hasR(obj1, steps, obj2)))`

The query method, as declared above, does not have access to the variables in the context of its call. To solve this problem, we pass such variables to the query method as additional parameters. For example, consider the expression `^(left + right + (a->b))`, where `left` and `right` are fields of a binary tree, and `a->b` denotes an added edge from node `a` to `b` (both program variables). When translated to JML, the parameters list of the query becomes: `Tree root, int steps, Tree node, Tree a, Tree b`.

We disallow taking the transitive closure of a relation that accesses both the pre- and post-state. This is because it is impossible to pass heap states to a JML model method. If the transitive closure relation accesses only the pre-state, we use the `\old` operator around the call to the query (see Fig. 3). Reflexive transitive closure is translated similarly and is described in [16]. Our Isabelle/HOL proofs do not cover rule R$_{37}$ because, unlike the rules of Table 1, R$_{37}$ requires a more elaborated formalization of JML and Java in order to express the semantics of the query method. Such a formalization could not be done as part of this work. Correctness of this rule has been manually validated instead.

4.3 The Modifies Clause

Each location in a `modifies` clause is given by the syntax `r.member`, where `r` is an Alloy expression that specifies the set of objects whose `member` field may be modified. Simple expressions (e.g. "`this.length`") can be directly translated into JML. In general, however, the object set cannot be expressed as a JML expression. In this case, Alloy2JML will generate a less specific JML modifies clause which allows modification of the `member` field of *all* objects rather than the ones specified by the expression `r`. In JML*, this is done using `\infinite_union`:

`//@ modifies \infinite_union(`*Type* `obj; obj.`*member*`);`

Furthermore, the translation generates an Alloy post-condition which specifies that the `member` field of any object not included in `r` remains unchanged:

`//$ ensures all v:`*Type* `-` *freshType* `- r | v.member'=v.member`

This post-condition is translated to JML as usual using the $\mathcal{B}$ function.

5 Evaluation

As a proof of concept that the generated JML specifications indeed are amendable to verification, we have applied Alloy2JML to 6 methods of two Java data structures: `constructor`, `add`, and `removeAt` of `LinkedList` and `constructor`,

(a) Alloy

```
1  /*$ensures this.*(left' + right').value' = (this.*(left + right).value) + v;
2     ensures (this.*(left' + right') - this.*(left + right)) in freshTree;
3     modifies this.*(left+right).left, this.*(left+right).right; */
```

(b) JML Translation

```
1  /*@ensures (\forall int o1;(\exists Tree o2,int i; i>=0 && hasLR(this,i,o2) && o2.value==o1)
2             <==> \old((\exists Tree o3,int j; j>=0 && hasLR(this,j,o3) && o3.value==o1)) || v == o1);
3  ensures (\forall Tree o1; (\exists int i; i>=0 && hasLR(this,i,o1)) &&
4            (\fresh(o1) || (\forall int j; j>=0 ==> !\old(hasLR(this,j,o1)))) ==> \fresh(o1));
5  ensures (\forall Tree o,int i;!\fresh(o)&&(i>=0==>!\old(hasLR(this,i,o)))==>o.left==\old(o.left));
6  ensures (\forall Tree o,int i;!\fresh(o)&&(i>=0==>!\old(hasLR(this,i,o)))==>o.right==\old(o.right));
7  modifies \infinite_union(Tree o;o.left), \infinite_union(Tree o;o.right); */
```

Fig. 4. Specification of the add method

`contains`, and `add` of `BinarySearchTree`. We have manually written the Alloy specifications, then automatically translated them to JML using Alloy2JML, and proved the resulting JML specifications using the KeY verification engine. The complete experiments are explained in [16] and can be found in `http://asa.iti.kit.edu/402.php`.

As an example, Fig. 4 shows the specifications of the `add` method of the class `BinarySearchTree`[5]. The method adds a node to a tree which is defined using its `left`, `right`, and `value` fields. `BinarySearchTree` also includes invariants to preserve sortedness and acyclicity of the tree. These invariants are omitted here due to lack of space. Given a value `v`, `add` recursively traverses the receiver tree, and inserts a new tree node containing `v` to the appropriate place if `v` is not already stored in the tree. The Alloy expression `this.*(left+right)` provides the set of all nodes reachable from the current node. Alloy's relational logic allows us to elegantly express the addition to the set of values in the tree nodes. In Fig. 4(a), Line 1 specifies that the values of the tree nodes in the post-state are the union of the nodes' values in the pre-state and the input argument `v`; Line 2 specifies that the nodes added to the tree are newly allocated objects. These two lines are translated respectively to Lines 1–2 and 3–4 of Fig. 4(b). Line 3 of Fig. 4(a) indicates that the memory locations referred to by the `left` and `right` fields (hereafter *locs*) of any node of the current tree can be changed by the method. Various translations of this `modifies` clause are possible. Alloy2JML translates this to Lines 5–7 of Fig. 4(b), which we found more amenable to verification. Lines 5–6 specify that the locs of any node that is not in the current tree stay unchanged. Line 7 specifies that the locs of any tree node can be changed by the method.

We used an experimental KeY version that has improved support for recursively specified query methods (e.g. `hasLR`). The query expansion and quantifier instantiation can be performed automatically in KeY. However, KeY may not always automatically find proofs. For any incomplete branch of the proof, we transformed the problem into first-order SMT logic that contains unbounded integers, uninterpreted functions and quantifiers, and tried to prove it using the

[5] In the interest of space, `BinarySearchTree` is named `Tree` in the figure.

Z3 SMT solver. If neither KeY nor Z3 could find an automatic proof, we manually performed explicit instantiations or query expansions, and provided several lemmas to assist KeY. In the case of `add`, the proof required 75 interactive steps, around 31000 automatic steps, and 40 subgoals were closed by Z3 invocations.

For the data structures that we analyzed, the Alloy specifications are concise (e.g. reachability via arbitrary combinations of fields is expressed easily, and frame conditions are implied elegantly). The generated JML specifications are readable, which is crucial for providing additional lemmas, and are provable using KeY.

To our knowledge, this is the first successful deductive verification of the operations on a tree data structure specified in standard JML. In [3], for example, a much bigger subset of JML* (including abstract data types and other features) is used to verify a remove operation of a tree. This subset, however, cannot be reduced to standard JML.

In order to check the compatibility of our generated JML specifications with JML tools other than KeY, we translated some of our JML* specifications to standard JML as explained in Section 2.2. All of our target JML tools, namely ESC/Java2 [6], JMLForge [8], InspectJ [22], TACO [12], and Krakatoa [10], accepted the resulting JML specifications.

6 Related Work

Several approaches, e.g. [5,17,19], translate the specification languages containing relations into JML. B2JML [5] presents a translation from B machines to JML specifications; in [19] a translation strategy from VDM-SL to JML is presented; and [17] provides a translation technique between OCL and JML. Unlike our approach, these approaches translate the relations of the source language to JML mathematical collections and the relational operators to JML model methods of those collections. We, on the other hand, translate relations to a basic variant of JML which can generally be used in other contexts after making the minor modifications described in Section 2.

JKelloy [14] translates Java programs annotated with Alloy specifications into the first order logic of KeY by defining a special relational theory in KeY. Similar to our approach, it enables full verification of Alloy specifications for Java programs. Alloy2JML, however, does not require any special background theory in the underlying verification engine, but provides a translation that can be used in other contexts as well.

TACO [11] and JMLForge [8] provide fully automated, bounded analysis of JML-annotated Java programs. These tools perform the reverse translation of what we do: they translate JML to a variant of Alloy by introducing the concept of method behavior of JML into Alloy. The resulting Alloy formula is then translated to a SAT problem, and solved using an off-the-shelf SAT solver.

A model transformation from a subset of Alloy to UML class diagrams annotated with OCL is presented in [13]. Their translation and simplifications have ideas common with ours, but their target domain is very different. In [9], we proposed a translation of Alloy to an SMT first-order logic by translating Alloy relations to membership predicates with set semantics. Here, on the other hand, we target Alloy expressions used as specifications of Java programs, and produce well-defined JML expressions (e.g. no null pointer dereferences) that respect the semantics of Java heap. Moreover, specializing the translation enables us to substantially improve the readability of the resulting JML expressions.

In [15], the JML query method `Node getNext(Node o, int n)` is manually specified to verify linked list data structure. The query provides access to the `n`'th node of the list starting from node `o`, following the field `next`. It complements the JML `reach` clause by additionally identifying the position of list nodes. Here we generalize that work by automatically generating the query method `hasR` (Section 4.2), which allows us to reason about arbitrary data structures.

7 Conclusion

JML is a popular specification language. Yet, manually specifying certain properties, e.g. those of linked data structures, can be complicated and error-prone when using a basic subset that is supported by most JML tools. On the other hand, Alloy operators (e.g., relational join, transitive closure, set comprehension, and set cardinality) let users concisely specify such properties. Hence we have built Alloy2JML, a tool that translates Alloy specifications to a basic subset of JML without the use of mathematical sets and containers. In most cases, we convert relational operators into JML first-order logic by quantifying over the elements of relations. For the transitive closure, we introduce recursively specified model methods. The outcome of the translation is suitable for verification and enabled us, among others, to verify methods of a tree class. Using Isabelle/HOL, we proved that our translation is correct for a subset of Alloy.

Alloy2JML also provides an Alloy model as output, thus the Alloy specifications of the code can also be validated using the Alloy Analyzer. Moreover, translating Alloy specifications into JML enables the use of Alloy specifications in a larger set of tools that accept only JML specifications.

Alloy2JML allows both Alloy and JML annotations to be used together, thus enabling to specify each property in the more appropriate language. Each annotation, however, must be written completely either in Alloy or in JML. We plan to design a uniform language that allows Alloy and JML subexpressions to be mixed in a wellformed manner. Such a combination has the potential to bring together the best of both paradigms. We also plan to add support for loop invariants, so that those too can be specified using the Alloy language.

References

1. K. Becker and G. T. Leavens. Class LinkedList. `http://www.eecs.ucf.edu/~leavens/JML-release/javadocs/java/util/LinkedList.html`.
2. B. Beckert, R. Hähnle, and P. H. Schmitt. *Verification of object-oriented software: The KeY approach.* Springer, 2007.

3. D. Bruns, W. Mostowski, and M. Ulbrich. Implementation-level verification of algorithms with KeY. *STTT*, pages 1–16, 2013.
4. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, R. Leino, and E. Poll. An overview of JML tools and applications. *STTT*, 7(3):212–232, 2005.
5. N. Cataño, T. Wahls, C. Rueda, V. Rivera, and D. Yu. Translating B Machines to JML Specifications. In *27th ACM Symp. on App. Comp.*, pages 1271–1277, 2012.
6. D. R. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *CASSIS*, volume 3362 of *LNCS*, pages 108–128, 2005.
7. G. Dennis, F. S.-H. Chang, and D. Jackson. Modular verification of code with SAT. In *ISSTA* , pages 109–120. ACM, 2006.
8. G. Dennis, K. Yessenov, and D. Jackson. Bounded verification of voting software. In *VSTTE*, volume 5295, pages 130–145. Springer, 2008.
9. A. A. El Ghazi and M. Taghdiri. Relational reasoning via SMT solving. In *FM*, pages 133–148, 2011.
10. J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV*, pages 173–177. Springer, 2007.
11. J. P. Galeotti, N. Rosner, C. Lopez Pombo, and M. Frias. Analysis of invariants for efficient bounded verification. In *ISSTA*, pages 25–36. ACM, 2010.
12. J. P. Galeotti, N. Rosner, C. G. L. Pombo, and M. F. Frias. TACO: Efficient SAT-based bounded verification using symmetry breaking and tight bounds. *IEEE Transactions on Software Engineering*, 39(9):1283–1307, Sept 2013.
13. A. G. Garis, A. Cunha, and D. Riesco. Translating Alloy specifications to UML class diagrams annotated with OCL. *SoSyM*, pages 1–21, 2013.
14. A. E. Ghazi, M. Ulbrich, C. Gladisch, S. Tyszberowicz, and M. Taghdiri. JKelloy: A proof assistant for relational specifications of Java programs. In *NFM*, 2014.
15. C. Gladisch and S. Tyszberowicz. Specifying a linked data structure in JML for formal verification and runtime checking. In *SBMF*, pages 99–114, 2013.
16. D. Grunwald. Translating Alloy specifications to JML. Master's thesis, Karlsruhe Institute of Technology, December 2013. http://asa.iti.kit.edu/410.php.
17. K. Hanada et al. Implementation of a prototype bi-directional translation tool between OCL and JML. *J. Informatics Society*, 5(2):89–95, 2013.
18. D. Jackson. *Software Abstractions (revised edition).* MIT Press, 2012.
19. D. Jin and Z. Yang. Strategies of Modeling from VDM-SL to JML. In *Advanced Language Processing and Web Information Technology*, pages 320–323, 2008.
20. G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.*, pages 619–695, 2006.
21. G. T. Leavens et al. JML Reference Manual (draft, revision 1.235), June 2008. `http://www.jmlspecs.org/`.
22. T. Liu, M. Nagel, and M. Taghdiri. Bounded program verification using an SMT solver: A case study. In *ICST*, pages 101–110, April 2012.
23. C. Marché et al. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *J. Log. Algebr. Program.*, 58(1–2):89–106, 2004.
24. P. Müller et al. Modular specification of frame properties in JML. *Concurrency and Computation: Practice and Experience*, 15(2):117–154, 2003.
25. M. Vaziri. *Finding Bugs in Software with a Constraint Solver.* PhD thesis, Massachusetts Institute of Technology, 2004.
26. B. Weiß. *Deductive Verification of Object-Oriented Software.* PhD thesis, Karlsruhe Institute of Technology, 2011.
27. K. T. Yessenov. A Lightweight Specification Language for Bounded Program Verification. Master's thesis, Massachusetts Institute of Technology, 2009.