Karlsruhe Reports in Informatics 2014,3
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

On Verifying Relational Specifications of Java Programs with JKelloy

Aboubakr Achraf El Ghazi, Mattias Ulbrich, Christoph Gladisch, Shmuel Tyszberowicz, and Mana Taghdiri

2014

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

On Verifying Relational Specifications of Java Programs with JKelloy *

Aboubakr Achraf El Ghazi, Mattias Ulbrich, Christoph Gladisch, Shmuel Tyszberowicz, and Mana Taghdiri

Karlsruhe Institute of Technology, Germany
{elghazi,ulbrich,christoph.gladisch,tyshbe,mana.taghdiri}@kit.edu

Abstract. Alloy 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. Several tools support Alloy specifications for Java programs. However, they can only check the validity of those specifications with respect to a bounded domain, and thus, in general, cannot provide correctness proofs. This paper presents JKelloy, a tool for deductive verification of Java programs with Alloy specifications. It includes automatically-generated coupling axioms that bridge between specifications and Java states, and two sets of calculus rules that (1) generate verification conditions in relational logic and (2) simplify reasoning about them. All rules have been proved correct. To increase automation capabilities, proof strategies are introduced that control the application of those rules. Our experiments on linked lists and binary graphs show the feasibility of the approach.

Keywords: first-order relational logic, relational specification, Alloy, Java, theorem proving, KeY

1 Introduction

The efficiency of specifying and verifying a linked data structure depends to a large extent on both the level of abstraction of that data structure and the conciseness of expressing a property over its reachable elements. A suitable formalism for expressing such properties that can also be utilized in the context of theorem proving is relational logic with a transitive closure operator. In this logic, the links of the data structures can be modeled as binary relations, and thus reachability can be expressed using transitive closure. Furthermore, relational specifications allow the user to easily abstract away from the exact order and connection of elements in a data structure by viewing it as a set. This reduction of precision, when applicable, pays off in simplification of proofs as well as in better readability of the specifications and the intermediate verification conditions, which is important for user interaction.

In this paper we describe JKelloy, our extension of the deductive Java verification tool KeY [3], to support specifications written in the relational specification

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

language Alloy [9]—a first-order relational logic with built-in operators for transitive closure, set cardinality, integer arithmetic, and set comprehension. To the best of our knowledge, this work is the first attempt in this direction; other related approaches either restrict the analysis to bounded domains (e.g. [1,6,18,20]) or focus only on the Alloy models of systems without considering their implementations (e.g. [2,14,17]). In our previous work [17] we formalized a translation from Alloy specifications into the KeY first-order logic, with the aim of full (i.e., *unbounded*) verification of declarative models of systems that are specified in Alloy. This, however, is not sufficient for handling Alloy as a specification language for Java programs since it has no explicit model of program state change.

JKelloy assumes a *relational view* of the Java heap: classes are modeled as Alloy signatures and fields as binary relations. To evaluate Alloy expressions in different program states, e.g. pre- and post-state of a method, we translate Alloy relations into functions which take the heap (representing the program state) as an argument. We define the relationship between Alloy relations and Java program states using pre-defined *coupling axioms*. This eliminates the need for the user to provide coupling invariants manually. Changes to program states are aggregated as heap expressions. We introduce an automatic transformation of those heap expressions to relational expressions using a set of *heap resolution rules* that normalize all intermediate heap expressions. The transformation allows us to reason about verification conditions in the relational logic. To simplify the reasoning process, we further introduce a set of *override simplification rules* that exploit the specific shape of the resulting conditions. To increase the degree of automation, we have developed two *proof strategies* that control the application of our rules. We have proved the correctness of all rules using KeY.

Given a Java program, JKelloy can also generate an *Alloy context* that maps the class hierarchy of the program to a semantically equivalent Alloy type hierarchy. This allows the user to check the consistency of the specifications using the automatic, lightweight Alloy Analyzer before starting the full, possibly interactive verification process. Building on top of KeY enables the user to take advantage of the supported SMT solvers to prove simpler subgoals. It also lets the user provide additional lemmas. Complex lemmas, e.g. those that contain transitive closure over update expressions, can be proved by using induction in side-proofs, and then be reused to automatically prove non-trivial verification conditions without requiring induction.

2 Overall Framework

Our verification tool JKelloy extends KeY [3], a deductive verification engine that supports both automatic and interactive verification of Java programs. Figure 1 presents the general structure of JKelloy as well as the user's workflow. The input of the tool is a Java program together with its specification written in Alloy [9]. JKelloy follows the *design-by-contract* [13] paradigm in which every method is specified individually with pre- and post-conditions. Verification is performed method by method, in a modular way. For simpler programs and properties, the

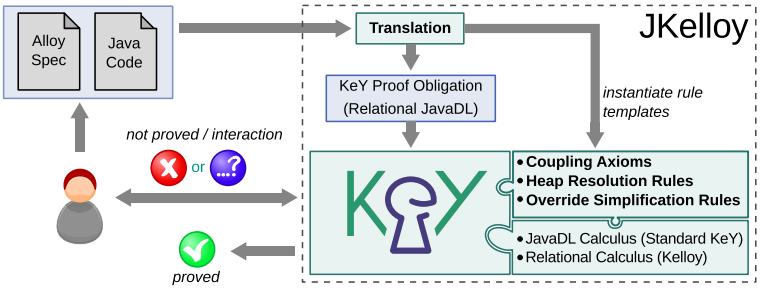

Fig. 1. Overall Framework. Contributions highlighted in a boldface font.

verification may run through automatically. In other cases, some user interaction may be required, in which the user guides the steps taken by the prover.

JKelloy extends KeY with a translation front-end that converts Alloy specifications of Java methods to *Java Dynamic Logic (JavaDL)*, the input logic of KeY. Our previous work, Kelloy [17], embedded general Alloy expressions into JavaDL (thus called *relational JavaDL*) and provided a basic relational calculus. JKelloy augments Kelloy with heap-dependent relations for modeling Java classes and fields. Furthermore, JKelloy introduces a set of calculus rules that facilitates verification of relational specifications. Some of these rules are program-dependent, and are generated for each program during the translation by instantiating pre-defined templates. The verification process for a method contract typically proceeds as follows:

1. The Alloy pre- and post-conditions are translated to relational JavaDL. The relations in the conditions become relational symbols depending on a heap-state. Their evaluation in a heap state is defined by *coupling axioms*.
2. The code of the Java method is symbolically executed, computing the post-heap-state in relation to the pre-heap.
3. *Heap resolution rules* are applied to normalize the resulting heap-dependent expressions so that all heap arguments become constant.
4. The resulting proof obligation is relational and can be discharged using the relational calculus. *Override simplification rules* simplify this process by providing additional lemmas in relational logic.

3 Alloy Specifications for Java Programs

Alloy [9] is a first-order relational logic with built-in operators for transitive closure, set cardinality, integer arithmetic, and set comprehension, which make it particularly suitable for concisely specifying properties of linked data structures. Properties of object-oriented programs can be specified in Alloy using the *relational view of the heap* [18]. That is, every class is viewed as a set of objects, and every field as a relation from the class in which the field is declared to its type.

```
1  class List {
2   Entry head;
3
4   void prepend(Data d) {
5    Entry oldHead = head;
6    head = new Entry();
7    head.next = oldHead;
8    head.data = d;
9   }
10 }
11
12 class Entry {
13  Data data;
14  Entry next;
15 }
16
17 interface Data {..}
18 class ID implements Data {..}
19 class Name implements Data {..}
```

(a)

```
1  one sig Null {}
2  sig Object' {}
3  sig Object in Object' {}
4  sig List' extends Object' {
5   head': one (Entry' + Null) }
6  sig List in Object {
7   head: one (Entry + Null) }
8  sig Entry' extends Object' {
9   data': one (Data' + Null),
10  next': one (Entry' + Null) }
11 sig Entry in Object {
12  data: one (Data + Null),
13  next': one (Entry + Null) }
14 sig ID' extends Object' {..}
15 sig ID in Object {..}
16 sig Name' extends Object' {..}
17 sig Name in Object {..}
18 sig Data' in Object' {..}
19 sig Data in Object {..}
20 fact { List = List' & Object
21         Entry = Entry' & Object
22         ID = ID' & Object
23         Name = Name' & Object
24         Data' = Name' + ID'
25         Data = Name + ID }
26 pred pre[self: one List, d: one (Data + Null)] {..}
27 pred post[self: one List, d: one (Data + Null)] {..}
```

(b)

Fig. 2. (a) Sample code (b) Alloy context

Our representation of this relational view differs from other approaches [18,20] in that it provides an *explicit* encoding of the Java types in the pre- and post-state.

Given a Java program, JKelloy automatically generates an *Alloy context* which encodes the type hierarchy of that program, and declares all the relations accessible to the user for writing the specifications. The user can then add the specifications to this context in order to check their consistency using the Alloy Analyzer before starting the verification process using JKelloy. Although the Alloy Analyzer checks Alloy models only for bounded domains, it helps users detect flaws automatically: under-specifications and errors can be detected using the visualizer tool in the Alloy Analyzer, whereas over-specification can be detected using the unsat-core generator tool.

Figure 2(a) provides a sample Java program. It implements a singly linked list that stores `Data` objects, where `Data` is declared as an interface with two sample implementations. The method `prepend` adds a `Data` object to the beginning of the list.

Figure 2(b) presents the corresponding Alloy context. A signature declaration `sig A{}` declares `A` as a top-level type (set of uninterpreted atoms); `sig B in A{}` declares `B` as a subtype (subset) of `A`. The `extends` keyword has the same effect as the keyword `in` with the additional constraint that extensions of a type are mutually disjoint. An attribute `f` of type `B` declared in signature `A` represents a relation $\texttt{f} \subseteq \texttt{A} \times \texttt{B}$. The multiplicity keyword `one`, when followed by a set,

constrains that set to be a singleton, and when used as a type qualifier of a relation, constrains that relation to be a total function.

The generated Alloy context always contains a singleton `Null` (Fig. 2(b) Line 1) which represents the Java `null` element. Every Java class `C` is represented by two signatures, `C` and `C'`, that give the set of atoms corresponding to the allocated objects of type `C` in the pre- and post-state, respectively. The top-level Java class `Object` is always included. The Alloy signature `Object` is constrained to be a subset of `Object'` (Line 3). This allows new objects to be created, but created objects cannot be deallocated. That is, garbage collection is not considered. If a Java class `B` extends a class `A` (immediate parent), the signature `B'` will be an extension of `A'`, and `B` a subset of `A`[1]. Furthermore, any pre-state signature `C` is constrained to be the intersection of its corresponding post-state signature `C'` and the signature `Object` (e.g. Lines 20–23). This ensures both $\texttt{C} \subseteq \texttt{C'}$ and $\texttt{C} \subseteq \texttt{Object}$. Signatures for interfaces denote the union of the signatures for the classes that implement them (Lines 24-25).

A Java field `f` of type `T` declared in a class `C` is represented by two functional relations $\texttt{f: C} \rightarrow \texttt{(T} \cup \texttt{Null)}$ for the pre-state, and $\texttt{f': C'} \rightarrow \texttt{(T'} \cup \texttt{Null)}$ for the post-state (e.g. Lines 5, 7). Since $\texttt{C} \subseteq \texttt{C'}$, the domain of `f'` includes `C` as well.

Pre-conditions of a method can access the receiver object (`self`) and that method's arguments. Post-conditions can additionally access the method's return value (`ret`) if any exists. These are given as parameters of the predicates `pre` and `post`, respectively (Lines 26-27). The user can copy the pre- and post-conditions of the analyzed method as bodies of these predicates, and check their consistency by providing Alloy assertions or by simply running the visualizer to see their satisfiable instances.

Specifications must be legal Alloy formulas. Basic formulas are constructed using subset (`in`) and equality (`=`) operators over Alloy expressions, and are combined using the usual logical connectives as well as universal (`all`) and existential (`some`) quantifiers. Alloy expressions evaluate to relations. Sets are unary relations and scalars are singleton unary relations. The operators `+`, `-`, and `&` denote union, difference, and intersection, respectively. For relations `r` and `s`, relational join (forward composition), Cartesian product, and transpose are denoted by `r.s`, `r -> s`, and `~r`, respectively. The relational override `r++s` contains all tuples in s, and any tuples of r whose first element is not the first element of a tuple in s. The transitive closure `^r` denotes the smallest transitive relation that contains `r`, and `*r` denotes the reflexive transitive closure of `r`. The expressions `s<:r` and `r:>s` give domain and range restriction of r to s, respectively.

We assume that pre- and post-conditions are annotations marked as `requires` and `ensures` clauses, respectively. Assume that the specifications of `prepend` read as follows:

[1] Semantically, the pre-state signature `B` must be an extension of the pre-state signature `A` rather than a subset. However, pre-state signature hierarchy goes up to the `Object` signature which is the subset of `Object'`, and Alloy does not allow subset signatures (in this case, `Object`) to have extensions. Nonetheless, it is easy to show that our other constraints imply subclasses of a class to be disjoint in the pre-state.

```
/*@ requires true;
  @ ensures self.head'.*next'.data'  = self.head.*next.data + d;
  @*/
```

In this case, the method has no pre-conditions, and the post-condition ensures that the set of `Data` objects stored in the receiver list in the post-state (given by the expression `self.head'.*next'.data'`) augments that of the pre-state (given by the expression `self.head.*next.data`) with the prepended data (namely `d`). This example shows that Alloy specifications for linked data structures tend to be concise. It also shows that the specifications can be arbitrarily partial.

4 Relational Java Dynamic Logic

4.1 Background

JavaDL, the verification logic of KeY, extends typed first-order logic with dynamic logic [8] operators over Java program fragments. Besides propositional connectives and first-order quantifiers, it introduces modal operators. The formula $\{p := t\}\varphi$ in which p is a constant symbol, t is a term whose type is compatible with that of p, and φ is a JavaDL formula, is true iff φ is true after the assignment of t to p. The modal operator $\{p := t\}$ is called an *update*. The formula $[\pi]\varphi$ in which π is a sequence of Java statements and φ is a formula, is true iff φ is true in the post-state (if any exists) of the program π. The formula $\langle\pi\rangle\varphi$ additionally requires π to terminate.[2]

JavaDL is based on an *explicit heap model* [19]: a dedicated program variable *heap* of type *Heap* stores the current heap state. A read access `o.f` in Java is encoded as $select(heap, o, \mathtt{f})$, abbreviated as $heap[o.f]$. Heap modifications are modeled using *heap constructors*, as defined in Fig. 3. The *store* function is used to encode changes to a field other than $\langle\mathtt{created}\rangle$. The boolean field $\langle\mathtt{created}\rangle$ is implicitly added to the class `Object` to distinguish between created and uncreated objects. A Java assignment of a variable v to a field `f` of a non-null object o can be interpreted as an update:

$$[\mathtt{o.f = v;}]\varphi \;\leftrightarrow\; \{heap := store(heap, \mathtt{o}, \mathtt{f}, \mathtt{v})\}\varphi \tag{1}$$

The *create* function is used to set the $\langle\mathtt{created}\rangle$ field of an object to **true**. The *anonymizing* function *anon* modifies a set of locations rather than a single location. It is used to summarize the effects on the heap made by code in loops or method invocations. The heap denoted by the term $anon(h_1, l, h_2)$ coincides with h_2 (the anonymous heap) in all fresh locations and those in the location set l, and coincides with h_1 (the base heap) on the remaining ones.

JavaDL's type system includes the hierarchy of Java reference types, with the root type *Object* which denotes an infinite set of objects (including the `null` object), whether or not created. The expression $free(h) = \{o : Object \mid$

[2] $[\pi]\varphi$ and $\langle\pi\rangle\varphi$ correspond to $wlp(\pi, \varphi)$ and $wp(\pi, \varphi)$ in the wp-calculus [4].

$$store(h,p,g,v)[o.f] = (\text{if } o = p \wedge f = g \wedge g \neq \langle \texttt{created} \rangle \text{ then } v \text{ else } h[o.f])$$
$$create(h,p)[o.f] = (\text{if } o = p \wedge f = \langle \texttt{created} \rangle \text{ then } \textbf{true} \text{ else } h[o.f])$$
$$anon(h_1,l,h_2)[o.f] = (\text{if } (o,f) \in l \wedge f \neq \langle \texttt{created} \rangle \vee o \in free(h_1) \text{ then } h_2[o.f] \text{ else } h_1[o.f])$$

Fig. 3. Definitions of heap constructors

$\neg h[o.\langle \texttt{created} \rangle] \wedge o \neq \texttt{null}\}$ gives the set of all uncreated objects of h. The types *Boolean* and *Integer* have their usual meanings, the type *Field* consists of all Java fields declared in the verified program, and *LocSet* consists of sets of locations, which are binary relations between *Object* and *Field*. For a type T, the type predicate $x \sqsubseteq T$ evaluates to true iff x is of type T.

KeY performs *symbolic execution* [10] of the given Java code. The effects of this execution on the program state are recorded as JavaDL updates. The equivalence (1), for instance, is used to encode the effect of the Java assignment `o.f=v`. Similar equivalences are used for other Java statements. Branching statements cause the proof obligation to split into cases; corresponding path conditions are assumed in each case. Consequently, symbolic execution resolves the original proof obligation $pre \rightarrow [p]post$ of a program p into a conjunction of formulas of the form $pre \wedge path \rightarrow \{\mathcal{U}\}post$, in which *path* stands for the accumulated path condition, and $\mathcal{U}$ for the accumulated state updates in an execution path.

In [17] we presented an embedding of Alloy into JavaDL (thus called *relational JavaDL*). This included new JavaDL types, namely *Atom* for elements of relations, and a Rel_n type for all n-ary relations (for each n). New function symbols for Alloy operators were introduced and defined using axioms. The integers in JavaDL were used to axiomatize transitive closure as it is not axiomatizable in pure first-order logic. We use $\cup$, $\setminus$, $\oplus$, $\times$, $\lhd$, `.`, *, $^+$ (ascending precedence order) to denote the symbols in relational JavaDL that correspond to the Alloy operators `+`, `-`, `++`, `->`, `<:`, `.`, `*`, `^`, respectively.

4.2 Coupling Axioms

The embedding of Alloy into relational JavaDL is not sufficient for verifying Java programs as it lacks a model of program state. To encode a relational view of the heap, we translate relations for Java classes and fields as heap-dependent function symbols. A Java class `C` is translated to a function symbol $C_{rel} : Heap \rightarrow Rel_1$ such that the expression $C_{rel}(h)$ gives the set of all created objects of type C in the heap h, as given by the first coupling axiom:

$$C_{rel}(h) := \{o \mid h[o.\langle \texttt{created} \rangle] \wedge o \sqsubseteq C \wedge o \neq \texttt{null}\} \tag{2}$$

It should be noted that, without loss of generality, we make *Atom* a supertype of *Object* to let Java objects be elements of relations as in Axiom 2. A Java field `f` of type `R` declared in a class `C` is translated to a function symbol $f_{rel} : Heap \rightarrow Rel_2$ where $f_{rel}(h)$ gives the set of all pairs (o_1, o_2) such that, in heap h, the created

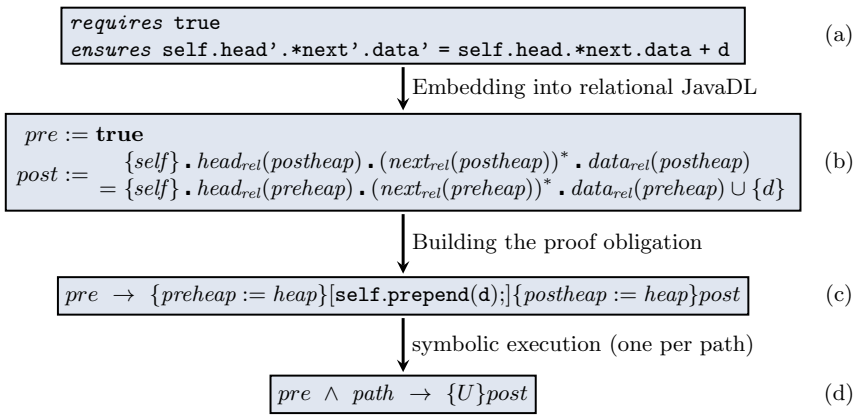

Fig. 4. The verification process for the method List.prepend as running example

object o_1 points to the object o_2 via `f`, as given by the second coupling axiom:

$$f_{rel}(h) := \{(o_1, o_2) \mid o_1 \in C_{rel}(h) \wedge (o_2 = \texttt{null} \vee o_2 \in R_{rel}(h)) \wedge o_2 = h[o_1.\texttt{f}]\} \quad (3)$$

Following the design-by-contract paradigm, Alloy specifications can access only the pre- and post-state. Thus we provide two sets of relations (unprimed for pre- and primed for post-state) instead of introducing an explicit notion of state. Heap arguments are introduced when Alloy specifications are translated into JavaDL: references to `C` and `f` are translated to $C_{rel}(preheap)$ and $f_{rel}(preheap)$, respectively, referring to the heap in the pre-state; references to `C'` and `f'` are translated to $C_{rel}(postheap)$ and $f_{rel}(postheap)$, referring to the heap in the post-state. `Null` signature is translated as $Null_{\text{rel}}(h) := \{\texttt{null}\}$ for every heap h.

Figure 4 shows how JKelloy processes the example of Fig. 2. Figure 4(a) is the original Alloy specification, Fig. 4(b) gives its translation into relational JavaDL, and Fig. 4(c) the relational JavaDL proof obligation for `List.prepend`. In addition to the program modality [`self.prepend(d);`], two updates $\{preheap := heap\}$ and $\{postheap := heap\}$ are used to store the respective current heap. Symbolic execution then resolves the code of the method. Several formulas of the form shown in Fig. 4(d) are produced for various execution paths of the code. The example is continued in Section 5.

The above coupling axioms are defined such that they preserve the meaning of the Alloy relations used in the specifications. For instance, relation `head'` in the example of Fig. 2 is a total binary relation containing the references from all created `List` objects to `Entry` objects (or `null`) after the method call. Axiom (3) ensures that $head_{rel}(postheap)$ contains precisely those elements.

5 Calculus

The coupling axioms (2) and (3) fix the semantics of the relation function symbols. Together with the relational calculus previously developed in Kelloy, they

suffice to conduct proofs for relational JavaDL formulas. In practice, however, this axiomatization is inefficient since it requires to always expand the definitions of the relations. In order to both lift proofs to the higher abstraction level of relations and to automate them, we introduce two sets of rules described in the following subsections.

5.1 Heap Resolution Rules

Figure 5 lists the rules for resolving heap constructor occurrences as argument of field relations (R_1–R_3) and class relations (R_4–R_6). All rules reduce relational expressions over composed heaps to expressions over their heap argument. They are applied to the verification conditions after symbolic execution and eliminate all heap constructors from arguments of relational function symbols. Rules R_1, R_2 and R_5, for instance, make case distinctions between the cases when the relation needs to be updated and when it remains untouched. R_3 is special since it updates a set of elements and not only one element in the relation. All rules were proved correct with respect to the coupling axioms.

We explain the idea of heap resolution using the example in Fig. 4. The update U in Fig. 4(d) encodes the successive heap modifications performed by the program. After some simplifications, the heap modification of the method body is encoded as

$$\begin{aligned} \mathit{postheap} := \overset{h_5}{\downarrow}\mathit{store}\,(\overset{h_4}{\downarrow}\mathit{store}\,(\overset{h_3}{\downarrow}\mathit{store}\,(\overset{h_2}{\downarrow}\mathit{create}\,(\overset{h_1}{\downarrow}\mathit{preheap}, e),\\ \mathit{self}, \texttt{head}, e),\\ e, \texttt{next}, \mathit{preheap}[\mathit{self}.\texttt{head}]),\\ e, \texttt{data}, d) \end{aligned}$$

where $h_1, \ldots, h_5$ are abbreviations for the intermediate heap expressions and e is a reference to the freshly created `Entry` object. In Fig. 4(b), some of the field relations take *postheap* as argument (like $\mathit{head}_{rel}(\mathit{postheap})$) in which, under the influence of U, *postheap* is replaced by the nested term h_5. Heap modifications in h_5 affect the value of $\mathit{head}_{rel}(h_5)$ only if they are related to the field `head`. Rule R_1, which is responsible for the resolution of this term, translates the *store* expression into an if-then-else term resulting either in an overridden relation ($f_{rel}(h) \oplus \{o_1\}\times\{o_2\}$) or in the original relation $f_{rel}(h)$. The relations $\mathit{head}_{rel}(h_5)$, $\mathit{head}_{rel}(h_4)$ and $\mathit{head}_{rel}(h_3)$, for instance, are equivalent as the modified Java fields `data` and `next` are different from `head`. But the *store* expression of h_3 modifies the field `head`; hence, the relation head_{rel} must be updated for the arguments of *store* and we obtain

$$\mathit{head}_{rel}(h_3) = \mathit{head}_{rel}(\mathit{store}(h_2, \mathit{self}, \texttt{head}, e)) = \mathit{head}_{rel}(h_2) \oplus \{\mathit{self}\}\times\{e\}\ . \quad (4)$$

Relation $\mathit{head}_{rel}(h_2)$ is finally simplified to $\mathit{head}_{rel}(h_1)$ by rule R_2 since the creation of the `Entry` element e does not affect the relation head_{rel} for the field `head` declared in class `List`. Equation (4) shows the main idea of the heap resolution calculus: the heap state changes are transformed into relational operations. In particular, an assignment `o1.f=o2` in Java resolves into a relational override of the form $f_{rel}(\mathit{heap}) \oplus \{o_1\}\times\{o_2\}$.

$\mathbf{R_1}$: $f_{rel}(store(h, o_1, g, o_2)) \rightsquigarrow$ if $g = \mathtt{f} \wedge h[o_1.created] \wedge o_1 \sqsubseteq C \wedge o_1 \neq \mathtt{null}$ then $f_{rel}(h) \oplus \{o_1\}\times\{o_2\}$ else $f_{rel}(h)$

assuming $wellformed(store(h, o_1, g, o_2))$

$\mathbf{R_2}$: $f_{rel}(create(h, o)) \rightsquigarrow$ if $o \neq \mathtt{null} \wedge o \sqsubseteq C$ then $f_{rel}(h) \oplus \{o\}\times\{h[o.\mathtt{f}]\}$ else $f_{rel}(h)$

$\mathbf{R_3}$: $f_{rel}(anon(h_1, l, h_2)) \rightsquigarrow f_{rel}(h_1) \oplus (((l\,.\,\{\mathtt{f}\}) \cup free(h_1)) \lhd f_{rel}(h_2))$

$\mathbf{R_4}$: $C_{rel}(store(h, o_1, g, o_2)) \rightsquigarrow C_{rel}(h)$

$\mathbf{R_5}$: $C_{rel}(create(h, o)) \rightsquigarrow$ if $o \neq \mathtt{null} \wedge o \sqsubseteq C$ then $C_{rel}(h) \cup \{o\}$ else $C_{rel}(h)$

$\mathbf{R_6}$: $C_{rel}(anon(h_1, ls, h_2)) \rightsquigarrow C_{rel}(h_1) \cup C_{rel}(h_2)$

Fig. 5. Heap Resolution Calculus. The term rewrite relation "$\rightsquigarrow$" represents an equivalence transformation. In R_1 and R_2 the field `f` is defined in class C.

Rule R_1 has been used three time in the above example. Twice (for h_5 and h_4) the condition $g = \mathtt{f}$ was false and the else branch had been taken. For h_3, the 4 parts of the condition were all true and the then-branch has been taken. For the soundness of R_1 with respect to the coupling axioms, it is required that the heap argument is *wellformed*, i.e. all locations point to a created object of their declared type. If this formula is not present in the verification condition, it is automatically introduced as a lemma by the strategy. When creating a new object o of class C in heap h and the field `f` is defined in C, rule R_2 extends the relation $f_{rel}(h)$ with a tuple for defining the value of $o\,.\,f_{rel}(h)$. Since the object o did not exist before, no such tuple was present in f_{rel}. Rule R_3 handles the *anon* constructor, where the memory locations l are assigned new values from h_2. Hence, the rule overrides the relation $f_{rel}(h_1)$ with tuples from $f_{rel}(h_2)$, but restricts this override to the relevant tuples. These are the locations in l that belong to field `f` (i.e. $l\,.\,\{\mathtt{f}\}$[3]) and the not yet created objects ($free(h_1)$). The selection is done using the domain restriction operator $\lhd$. Rule R_4 is relatively simple since the constructor *store* cannot modify the $\langle\mathtt{created}\rangle$ field, the set of created objects cannot be enlarged. Rule R_5 extends the relation C_{rel} when a new object of class `C` is created. Rule R_6 considers the possibility that *anon* introduces new objects, of any class, from h_2. The set of created objects in the *anon* heap contains all created objects of the first heap argument (objects can never be deallocated) and all objects created in the second (which have been introduced in the anonymization).

Applying these rules exhaustively leads to a normal form where all heap arguments are constants. JKelloy extends KeY with a proving strategy that always achieves this task automatically. The final result of applying the heap resolution rules to the post-condition of the running example (Fig. 4(b)) is the following

[3] To unify the presentation we apply relational operators also to type *LocSet*.

$\mathbf{R}_7$: $\{a\} \,.\, (R \oplus \{a\} \times \{b\}) \rightsquigarrow \{b\}$

$\mathbf{R}_8$: $S_1 \,.\, (R \oplus S_2 \times S_3) \rightsquigarrow$ if $S_2 = \emptyset$ then $S_1 \,.\, R$ else $S_1 \,.\, (S_2 \times S_3) \cup (S_1 \setminus S_2) \,.\, R$

$\mathbf{R}_9$: $S_1 \,.\, (R \oplus \{a\} \times S_2)^+ \rightsquigarrow$ if $S_2 = \emptyset \vee a \notin S_1$ then $S_1 \,.\, R$
else $S_2 \cup ((S_2 \setminus \{a\}) \,.\, R^+) \cup (S_1 \,.\, R^+)$
assuming $R \,.\, \{a\} = \emptyset$

$\mathbf{R}_{10}$: $\{a\} \,.\, (R \oplus \{b\} \times \{c\})^+ \rightsquigarrow$ if $b \in \{c\}.R^+ \vee b = c$ then $(\{a\} \,.\, R^+ \cup \{c\} \cup \{c\} \,.\, R^+) \setminus \{b\} \,.\, R^+$
else $(\{a\} \,.\, R^+ \setminus \{b\} \,.\, R^+) \cup \{c\} \cup \{c\} \,.\, R^+$
assuming $b \in a \,.\, R^+$, $parFun(R)$ and $acyc(R)$

$\mathbf{R}_{11}$: $\{a\} \,.\, (R \oplus \{b\} \times \{c\})^+ \rightsquigarrow$ if $b \neq a$ then $\{a\} \,.\, R^+$
elseif $a \in \{c\} \,.\, R^+ \vee c = a$ then $(\{c\} \cup \{c\} \,.\, R^+) \setminus \{a\} \,.\, R^+$
else $\{c\} \cup \{c\} \,.\, R^+$
assuming $b \notin a \,.\, R^+$ and $parFun(R)$

$\mathbf{R}_{12}$: $S_1 \,.\, f_{rel}(h) \,.\, (R_1 \oplus S_2 \lhd R_2) \rightsquigarrow S_1 \,.\, f_{rel}(h) \,.\, R_1$ assuming $S_2 \subseteq free(h)$

$\mathbf{R}_{13}$: $S_1 \,.\, f_{rel}(h) \,.\, (g_{rel}(h) \oplus S_2 \lhd R)^+ \rightsquigarrow S_1 \,.\, f_{rel}(h) \,.\, g_{rel}(h)^+$ assuming $S_2 \subseteq free(h)$

$\mathbf{R}_{14}$: $(f_{rel}(h) \oplus S_2 \lhd R)^+ \rightsquigarrow f_{rel}(h)^+ \oplus S_2 \lhd R$ assuming $S_2 \subseteq free(h)$

Fig. 6. A sampling of our override driven calculus rules

relational verification condition:

$$\begin{aligned} &\{self\} \,.\, (head_{rel}(h_1) \oplus \{self\}\times\{e\}) \\ &\quad \,.\, (next_{rel}(h_1) \oplus \{e\}\times\{self\} \,.\, head_{rel}(h_1))^* \,.\, (data_{rel}(h_1) \oplus \{e\}\times\{d\}) \\ = \; &\{self\} \,.\, head_{rel}(h_1) \,.\, next_{rel}(h_1)^* \,.\, data_{rel}(h_1) \cup \{d\} \end{aligned} \tag{5}$$

After all heap terms have been resolved, further reasoning can proceed on the relational level.

5.2 Override Simplification Rules

The normalized proof obligations that result from applying heap resolution rules can be proved on the relational level using our previous Kelloy tool. However, Kelloy only provides definition axioms for relational operators and a set of lemmas for general relational expressions. To make proofs easier and to increase the automation level, we introduce a set of lemma rules which exploit the shape of the relational expressions that result from verifying Java programs. These lemmas do not increase the power of the calculus but ease the verification by reducing the need for expanding the definitions of relational operators. That is particularly costly for the transitive closure as it leads to quantified integer formulas that generally require user interaction in form of manual induction. Out of more than 220 new lemmas we have introduced, we present the subset that is most relevant to the examples of Fig. 2 and Section 6; not all of them are used in the presented examples. All lemmas have been proved correct using KeY.

Equation (5) is typical for our approach: its right-hand side (RHS) refers to the base relations of the pre-state, whereas its left-hand side (LHS) refers to the

post-state and thus includes override-updates on the field relations. To prove such formulas, we bring the LHS closer to the shape of the RHS by resolving or pulling out the override operations that occur below other operators such as join and transitive closure.

Figure 6 lists a number of lemmas dealing with this override resolution to give an idea of the process. The most simple case is R_7 which says that retrieving a value a from a relation which has been overridden at the very same a results precisely in the updated value b. In other, more composed cases, the resolution is not as simple. Rules R_9, R_{10} and R_{11}, e.g., allow us to resolve the override beneath a transitive-closure operation under certain conditions at the cost of larger replacement expressions without override. Rules R_{12}–R_{14} resolve override operations which only modify objects not yet created in the base heap ($S_2 \subseteq \mathit{free}(h)$).

In the example, the subexpression $\{\mathit{self}\} \,.\, (\mathit{head}_{\mathit{rel}}(h_1) \oplus \{\mathit{self}\}\times\{e\})$ in (5) can be simplified to $\{e\}$ using R_7 as the left argument $\{\mathit{self}\}$ of the join equals the domain of the overriding relation $\{\mathit{self}\}\times\{e\}$. After this simplification, the LHS contains the subexpression

$$\{e\} \,.\, (\mathit{next}_{\mathit{rel}}(h_1) \oplus \{e\}\times\{\mathit{self}\} \,.\, \mathit{head}_{\mathit{rel}}(h_1))^* \ .$$

To resolve the override operation in this expression, we first transform reflexive transitive closure to transitive closure using the equality $S.R^* = S \cup S.R^+$, and then apply rule R_9. The assumption of R_9 holds because e is not yet created in h_1, and the if-condition evaluates to **false**. Further simplifications on the else-expression of R_9 based on the fact that e is uncreated in h_1 result in:

$$\{e\} \cup \underline{\{\mathit{self}\} \,.\, \mathit{head}_{\mathit{rel}}(h_1) \cup \{\mathit{self}\} \,.\, \mathit{head}_{\mathit{rel}}(h_1) \,.\, \mathit{next}_{\mathit{rel}}(h_1)^+}$$

The underlined subexpression is equivalent to $\{\mathit{self}\} \,.\, \mathit{head}_{\mathit{rel}}(h_1) \,.\, \mathit{next}_{\mathit{rel}}(h_1)^*$ which also appears on the RHS of (5). We have thus reached our goal of resolving the override and bringing the LHS closer to the RHS.

The rules of Fig. 6 work as follows. Rule R_8 is a generalization of R_7 (explained above) where an arbitrary relation is joined with an overridden expression. Pulling override out of a transitive closure operation is particularly important due to the complexity of verifying transitive closure. We introduce a number of rules for various forms of such expressions. Rule R_9, e.g., is applicable when the singleton in the domain of the overriding relation (a) is not in the range of the overridden relation (R)[4]. Rule R_{10}, which is one of the most general cases that our calculus can handle, is applicable under three assumptions: (1) the first element of the overriding pair (b) must be reachable from the joining singleton (a) via R, (2) the overridden relation (R) must be a partial function, (3) R must be be acyclic. Rule R_{11} complements R_{10}. It handles the case where the first element of the overriding pair (b) is *not* reachable from the joining singleton (a) via R. In this case, our rule is more general than the previous case since it does not require acyclicity of the overridden relation. Rules R_{12}–R_{14} resolve override operations which only modify the uncreated objects in the base heap

[4] Such information is inferred from the path condition in the proof obligation.

$\mathbf{R_{15}}$: $\vdash parFun(f_{rel}(h))$
$\mathbf{R_{16}}$: $\vdash parFun(R) \rightarrow parFun(R \oplus \{a\} \times \{b\})$
$\mathbf{R_{17}}$: $\vdash parFun(R_1) \wedge parFun(R_2) \rightarrow parFun(R_1 \oplus R_2)$

$\mathbf{R_{18}}$: $\vdash acyc(R) \wedge R \,.\, \{a\} = \emptyset \wedge a \neq b \rightarrow acyc(R \oplus \{a\} \times \{b\})$
$\mathbf{R_{19}}$: $\vdash acyc(R) \wedge \{b\} \,.\, R = \emptyset \wedge a \neq b \rightarrow acyc(R \oplus \{a\} \times \{b\})$
$\mathbf{R_{20}}$: $\vdash acyc(R) \wedge a \notin \{b\} \,.\, R^+ \wedge a \neq b \rightarrow acyc(R \oplus \{a\} \times \{b\})$

$\mathbf{R_{21}}$: $S_2 \in S_1 \,.\, R^+ \rightsquigarrow \mathbf{false}$ assuming $S_1 \,.\, R = \emptyset$
$\mathbf{R_{22}}$: $\{a\} \in R \,.\, \{b\} \rightsquigarrow \mathbf{true}$ assuming $\{a\} \,.\, R = \{b\}$
$\mathbf{R_{23}}$: $\{a\} \in R \,.\, \{b\} \rightsquigarrow \mathbf{false}$ assuming $parFun(R)$ and $\{a\} \,.\, R \neq \{b\}$

Fig. 7. A selection of auxiliary rules for the override simplification

($S_2 \subseteq free(h)$). Such cases arise when occurrences of the *anon* constructor are resolved by applying rule R_3.

The simplification rules focus on resolving override operations, yet further rules are required to reason about expressions that occur in the rules' assumptions, if-conditions, and results. Fig. 7 shows such rules divided into three categories. The first involves partial functionality of relations: every relation corresponding to a field is a partial function by construction (R_{15}); R_{16} and R_{17} allow the propagation of this property over the override operator. Similarly, the second propagates the acyclicity of relations over the override operator. The last category lists some rules for handling reachability between objects effectively.

The general shape of the Alloy expressions which our override-driven calculus can effectively simplify (i.e., their override operators can be pulled out to the top) is described in the grammar of Fig. 8. The fragment has three main restrictions: (1) override expressions are built by successively applying one or more override operations to field relations in which every overriding relation contains at most one element, (2) (reflexive) transitive closure operator can be applied to override expressions only if the left subexpression of the outermost override is acyclic, (3) override expressions may be joined from left only with a set that has at most one element.

The heap resolution rules resolve assignments to heap locations into override expressions for which the first restriction holds. These expressions have exactly the same form as $overE_1$ in Fig. 8. Anonymised heaps (using *anon*) do not belong to this fragment. The second and third restrictions are important in order to develop efficient resolution rules for override. Acyclicity is a property that is often assumed in linked data structures. Using the calculus rules, e.g. R_{18}-R_{20}, the acyclity of override expressions can be deduced from the acyclity of their base field relations. Specifications adhere to the third restrictions if they denote sets of objects reachable from a particular starting point (like the post-condition in Fig. 4(a) for instance). However, this last restriction does not hold for general relations which may also appear in specifications. The weakening of the last restriction is left for future work.

$expr$ ::= **self** | **var** | **none** | **univ** | **iden** | $\mathbf{C_{rel}(h)}$ | $\mathbf{f_{rel}(h)}$ | $overFreeE(^{\sim} \mid ^{+} \mid ^{*})$
| $overFreeE\ binOp\ overFreeE$ | $overE\ binOp\ overFreeE$ | $overFreeE\ binOp\ overE$
$overFreeE$ ::= `any override free expression`
$overE$ ::= $loneS \mathbin{.} overE_1$ | $loneS \mathbin{.} overE_2$ | $loneS \mathbin{.} (overE \cup overE)$
$overE_1$ ::= $\mathbf{f_{rel}(h)} \oplus loneS \times loneS$ | $overE_1 \oplus loneS \times loneS$
$overE_2$ ::= $(overE_1 \oplus loneS \times loneS)(^{+} \mid ^{*})$ `where acyc(`$overE_1$`)`
$loneS$ ::= **none** | **{self}** | **{var}** | $loneS \mathbin{.} \mathbf{f_{rel}(h)}$
$binOp$ ::= $\cup \mid \cap \mid \setminus \mid \times \mid \lhd \mid \rhd \mid .$

Fig. 8. Target fragment of the override driven calculus

6 Evaluation

Proofs in KeY are conducted by applying calculus rules either manually or automatically, using KeY's proof search strategy. We extend the existing strategy by incorporating two new strategies that assign priorities to heap resolution rules and override simplification rules, and apply them consecutively. The `List.prepend` example[5] verifies fully automatically within 5.4 seconds[6] using 1546 rule applications although its post-condition involves transitive closure.

We have also verified a slightly different example (`List.append`) where the `Data` argument is added to the end of the list. The proof contains a total of 2850 rule applications out of which 28 are interactive. These include 6 applications of proof-branching rules, and 6 rule applications to establish the assumptions for rule R_{10}. Automatic rule applications take 20.3 seconds. The `append` method is more complex than `prepend` as it contains a loop that traverses the list to the end, thus requires handling loop invariants. The proof requires the more complex transitive closure rule R_{10} since the code updates already-created objects.

We illustrate that JKelloy can be used to verify programs which manipulate rich heap data structures using the example of Fig. 9. This example also illustrates that structurally complex specifications can be concisely expressed by exploiting combinations of relational operators in Alloy. The `Graph` class implements a binary graph[7] where each node stores its two (possibly `null`) successors (`left` and `right`, Line 25). The graph keeps a linked list of its nodes (Line 2) using the `next` field (Line 25). The method `Graph.remove` removes a node `n` from the receiver graph by removing all of its incoming edges (Lines 17–21), and then removing `n` (and thus its outgoing edges) from the node list (Line 22).

The method requires the node list to be acyclic (Line 3) and the argument node `n` to be non-null (Line 4). It ensures that `n` is removed from the graph's node list (Line 5), and that the `left` and `right` fields of all nodes in this list that used to point to `n`, point to `null` at the end of the method (Lines 6 and 7). This example also illustrates that structurally complex specifications can be concisely expressed by exploiting combinations of relational operators in Alloy.

[5] All examples and proofs can be found at `http://i12www.ira.uka.de/~elghazi/jkelloy/`
[6] On an Intel Core2Quad, 2.8GHz with 8GB memory
[7] A directed graph with an outgoing degree of at most two for every node

```
1  public class Graph {
2    NodeList nodes;
3    /*@ requires acyc(next);
4      @ requires not n = null;
5      @ ensures self.nodes'.first'.*next' = self.nodes.first.*next - n;
6      @ ensures Object <: left' = left ++ ((left.n & self.nodes.first.*next) -> null);
7      @ ensures Object <: right' = right ++ ((right.n & self.nodes.first.*next) -> null); @*/
8    void remove(Node n) {
9      if (nodes != null) {
10       Node curr = nodes.first;
11       /*@ loop_invariant
12         @   curr in self.nodes.first.*next and
13         @   Object<:left' = left ++ ((left.n & (self.nodes.first.*next - curr.*next)) -> null) and
14         @   Object<:right' = right ++ ((right.n & (self.nodes.first.*next - curr.*next)) -> null)
15         @ assignable
16         @   (self.nodes.first.*next -> left) + (self.nodes.first.*next -> right); @*/
17       while (curr != null) {
18         if (curr.left == n)  { curr.left = null; }
19         if (curr.right == n) { curr.right = null; }
20         curr = curr.next;
21       }
22       nodes.remove(n);
23 } } }
24 class NodeList { Node first;  void remove(Node n) { ... } }
25 class Node {  Node next, left, right; }
```

Fig. 9. Specification and implementation of the graph remove example

In particular, sets of nodes with a particular property can be easily expressed using Alloy operators. For example, using the join operator from the right side of a field relation, the expression `left.n` concisely gives the set of all nodes whose `left` field points to `n`. The domain restriction to `Object` restricts the relation in the post-state to those objects already existing in the pre-state. The relational override operator denotes exactly what locations are modified and how, thus also implicitly specifies which locations do not change.

The example requires additional intermediate specifications which are not part of the contract. This includes a *loop specification* (Lines 11–16) describing the state after the execution up to the current loop iteration. Primed relations in the loop invariant refer to the state of the heap after the current loop iteration, whereas unprimed relations refer to the pre-state of the method. The assignable clause specifies the set of heap locations which may be modified by the loop. `Graph.remove` calls `NodeList.remove` which removes `n` from the linked list; the call is abstracted by the callee's contract which is omitted here for space reasons.

Though the specification in the example is concise, it extensively combines relational operators including, in particular, transitive closure. In the code, the nested method call and the loop result in complex composed heap expressions after symbolic execution. Brought together, these two technical points make this example difficult to verify. The proof required 6973 rule applications distributed over 157 subgoals, where 1201 of the rule applications were interactive. Amongst them, 309 apply override simplification rules and 224 general relational rules. Our rules for handling transitive closure proved to be very effective; they were applied 43 times, and allowed us to conduct the proof without any explicit induc-

tion. Induction was needed only to prove the soundness of the rules themselves. Relational operations were never expanded to their definitions. Thus the proof was completely conducted in the abstraction level of relations. The rules introduced with JKelloy made up 37% of all rule applications; the rest were default KeY rules. The whole proof, including specification adjustments, was conducted by an Alloy and KeY expert in one week; the total time spent by the automatic rule applications was 6.3 minutes. Other comparable examples in KeY (using the JML specification language) require $50k$ to $100k$ proof steps (see e.g. [7]).

7 Related Work

Several approaches (e.g. [5, 16, 18]) support Alloy as a specification language for Java programs. To check the specifications, however, they bound the analysis domain by unrolling loops and limiting the number of elements of each type. Thus although they find non-spurious counterexamples automatically, they cannot, in general, provide correctness proofs. JForge specification language [20] is another lightweight language for specifying object-oriented programs. It is a behavioral interface specification language with a relational view of the heap, that allows some Alloy operators. So far it has been used only for bounded program checking.

Galeotti [6] introduced a bounded, automatic technique for the SAT-based analysis of JML-annotated Java sequential programs dealing with linked data structures. It incorporates (i) DynAlloy [1], an extension of Alloy to better describe dynamic properties of systems using actions, in the style of dynamic logic; (ii) DynJML, an intermediate object-oriented specification language; and (iii) TACO, a prototype tool which implements the entire tool-chain.

A few approaches [2, 14, 17] support full verification of Alloy models. Since they do not model program states, they cannot be readily applied for verifying code with Alloy specifications. `DYNAMITE` [14], for example, extends PVS to prove Alloy assertions, and incorporates Alloy Analyzer for checking hypotheses.

Other approaches (e.g. [7, 15, 21]) also verify properties of linked data structure implementations. In contrast to ours, in [21], for example, specifications are written in classical higher-order logic (including set comprehension, λ-expressions, transitive closure, set cardinality) and are verified using Jahob which integrates several provers. A decision procedure based on inference rules for a quantifier-free specification language with transitive closure is presented in [15]. In [7] the focus is to write specifications in JML so that they can be used for both deductive program verification and runtime checking.

Similar to our approach, [11,12] handle reachability of linked data structures using a first-order axiomatization of transitive closure. Their general idea, however, is to use a specialized induction schema for transitive closure, to provide useful lemmas for common situations. [11] focuses on establishing a relatively complete axiomatization of reachability, whereas [12] focuses on introducing as complete schema lemmas as possible and adding their instantiations to the original formula. The main difficulty of schema rules is to find the right instantiation (analogous to induction hypothesis).

8 Conclusions

We have presented an approach for verifying Java programs annotated with Alloy specifications. Alloy operators (e.g. relational join, transitive closure, set comprehension, and set cardinality) let users specify properties of linked data structures concisely. Our tool, JKelloy, translates Alloy specifications into relational Java Dynamic Logic and proves them using KeY. It introduces coupling axioms to bridge between specifications and Java states, and two sets of calculus rules and strategies that facilitate interactive and automatic reasoning in relational logic. Verification is done on the level of abstraction of the relational specifications. JKelloy lets relational lemmas be proved beforehand, and reused to gain more automation. Our calculus rules are proved lemmas that exploit the shape of the relational expressions that occur in proof obligations.

Although our automatic proof strategies can still be improved, our examples show the advantages of the approach. They illustrate how the liberal combinations of transitive closure and relational operators in Alloy can be exploited for concise specifications of linked data structures. The sizes of proofs are an order of magnitude smaller compared to other similar proofs using standard KeY.

KeY supports JML, a behavioral specification language for Java. A combination of the specification concepts of JML and Alloy has the potential to bring together the best of both paradigms. Furthermore, the symbolic execution engine of KeY along with our calculus rules can produce relational summaries of Java methods which can be checked for bugs using the Alloy Analyzer before starting a proof attempt. Investigating these ideas is left for future work.

References

1. N. Aguirre, M. F. Frias, P. Ponzio, B. J. Cardiff, J. P. Galeotti, and G. Regis. Towards abstraction for DynAlloy specifications. In *ICFEM*, pages 207–225, 2008.
2. K. Arkoudas, S. Khurshid, D. Marinov, and M. Rinard. Integrating model checking and theorem proving for relational reasoning. In *RelMiCS*, pages 21–33, 2003.
3. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach.* Springer-Verlag, 2007.
4. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, Aug. 1975.
5. J. Dolby, M. Vaziri, and F. Tip. Finding bugs efficiently with a SAT solver. In *FSE*, pages 195–204, 2007.
6. J. P. Galeotti. *Software Verification using Alloy.* PhD thesis, Universidad de Buenos Aires, 2010.
7. C. Gladisch and S. S. Tyszberowicz. Specifying a linked data structure in JML for formal verification and runtime checking. In *SBMF*, pages 99–114, 2013.
8. D. Harel, J. Tiuryn, and D. Kozen. *Dynamic Logic.* MIT Press, 2000.
9. D. Jackson. *Software Abstractions: Logic, Language, and Analysis.* MIT Press, 2012.
10. J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

11. S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *Proceedings of POPL*, pages 115–126. ACM, 2006.
12. T. Lev-Ami, N. Immerman, T. W. Reps, M. Sagiv, S. Srivastava, and G. Yorsh. Simulating reachability using first-order logic with applications to verification of linked data structures. *Logical Methods in Computer Science*, 5(2), 2009.
13. B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.
14. M. Moscato, C. Lopez Pombo, and M. Frias. Dynamite 2.0: New features based on UnSAT-core extraction to improve verification of software requirements. In *ICTAC*, pages 275–289, 2010.
15. Z. Rakamarić, J. Bingham, and A. J. Hu. An inference-rule-based decision procedure for verification of heap-manipulating programs with mutable data and cyclic data structures. In *VMCAI*, pages 106–121, 2006.
16. M. Taghdiri. *Automating Modular Program Verification by Refining Specifications*. PhD thesis, MIT, 2008.
17. M. Ulbrich, U. Geilmann, A. A. El Ghazi, and M. Taghdiri. A proof assistant for Alloy specifications. In *TACAS*, pages 422–436, 2012.
18. M. Vaziri. *Finding Bugs in Software with Constraint Solver*. PhD thesis, MIT, 2004.
19. B. Weiß. *Deductive Verification of Object-Oriented Software: Dynamic Frames, Dynamic Logic and Predicate Abstraction*. PhD thesis, KIT, 2010.
20. K. T. Yessenov. A Lightweight Specification Language for Bounded Program Verification. Master's thesis, MIT, 2009.
21. K. Zee, V. Kuncak, and M. Rinard. Full functional verification of linked data structures. In *PLDI*, pages 349–361, 2008.