Karlsruhe Institute of Technology

Generating Bounded Counterexamples for KeY Proof Obligations

Master Thesis of

Mihai Herda

At the Department of Informatics
Institute of Theoretical Computer Science

Reviewer: JProf. Dr. Mana Taghdiri
Prof. Dr. Bernhard Beckert

Advisors: Aboubakr Achraf El Ghazi
Mattias Ulbrich
Christoph Gladisch

Time Period: 4th July 2013 – 3rd January 2014

Statement of Authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Karlsruhe, 19th December 2013

Acknowledgements

I would like to thank my supervisors, Aboubakr Achraf El Ghazi, Christoph Gladisch, Mana Taghdiri, and Mattias Ulbrich for the weekly meetings which helped me better understand the tasks and problems at hand. I would also like to thank Daniel Bruns for his opinions on the tool. Last but not least, I would like to thank my family and my friends for their continuous and unconditional support.

Abstract

KeY is an interactive software verification system which can verify Java programs specified with JML. It uses a sequent calculus for a dynamic logic for Java. KeY supports automation to a certain degree, but when user interaction is required it is difficult to determine whether a proof obligation is invalid. For this reason, we designed and implemented a tool for finding counterexamples for KeY proof obligations. It works by translating the negation of a KeY proof obligation to an SMT specification, with all SMT sorts bounded, thus ensuring decidability. This translation is then handed over to an SMT solver. We make sure that interpreted KeY functions and predicates, preserve their semantics in order to avoid spurious counterexamples caused by the loss of their semantics. We also preserve the KeY type hierarchy. Additionally we make sure that integer overflows are not used in the found counterexamples. Because the output of the SMT solver is difficult to read, we extract the relevant information from it and present it in a user friendly manner. We have evaluated our tool on both faulty and fault free specified Java programs, and showed how the tool can be used to understand why a proof obligation is not valid.

Contents

List of Figures

List of Tables

1. Introduction

1.1. Motivation

KeY is a software verification system which can prove that Java programs fulfill their specification. It uses JavaDL, a dynamical logic for reasoning about Java programs. The reasoning is done with a sequent calculus for JavaDL, which works by applying syntactic rules on proof obligations. Rules can generate new proof obligations or close them. The objective is to close all proof obligations. If that is achieved, the validity of the original JavaDL formula is proven. KeY can automatically apply rules according to its own heuristics. If the automatic rule application fails, the user is shown the proof obligations, where the proof got stuck. The automatic rule application can fail because the initial proof obligation is not valid, or because user interaction is required in order to advance the proof. It is often difficult for to user to determine, which one of these two causes apply. This dilemma provides the main motivation for our work.

An additional motivation is given by the fact that KeY can only prove the validity of JavaDL formulae, but it is not able to provide counterexamples for invalid formulae. This can make it hard for the user to understand why a formula is not valid.

1.2. Project Goal

The goal of this project is to design and implement a tool for finding counterexamples for KeY proof obligations using an SMT solver.

The tool receives a proof obligation as an input, and will try to show that the proof obligation is not valid by providing a counterexample. Should the tool succeed, the user will know that the proof obligation, and thus the entire proof, is not closable. Additionally, the counterexample can help him understand the reason for the failure.

The tool works by translating a proof obligation to the SMT-LIB language. Because the specification for SMT solvers is written in typed first order logic with additional theories, we can only support proof obligations written in KeY first order logic (KeYFOL), which is a subset of JavaDL. However, KeY is able to automatically apply the necessary rules in order to obtain only proof obligations written in KeYFOL. The translated proof obligation is negated and given to an SMT solver. A model for the negated proof obligation is a counterexample for the original proof obligation.

All KeY types are translated as bounded SMT sorts. This means, that there are finitely many instances of every SMT sort. As a result all SMT specifications we obtain are decidable. It is possible, however, for the SMT solver to not have enough physical resources at its disposal and time out. In this case, the user can try smaller bounds.

KeY already can translate proof obligations to SMT, but this translation serves the purpose of proving proof obligations. For proving it is not necessary to translate the semantics of KeY predicates and functions and many are left underspecified. However, for counterexample finding the KeY functions and predicates cannot be underspecified, otherwise the SMT solever will wrongly define them, rendering the found counterexample spurious.

Because unbounded KeY types are translated to bounded SMT sorts, there are proof obligations for which our tool will always find spurious counterexamples. We accept this fact as unavoidable, and try to minimize the causes of spurious counterexamples. We achieve this by providing an accurate translation for KeY functions, predicates, and type hierarchy, and by adding formulae which disallow integer overflows. The only proof obligations which produce spurious counterexamples are those which require some types to be unbounded in order to be valid. In practice, however, such proof obligations rarely result from proving properties of specified Java programs, which is the main use case of KeY.

1.3. Outline

Chapter 2 presents JavaDL and a sequent calculus for JavaDL as well as the KeY system. Furthermore it introduces SMT concepts that are needed to understand the translation. Chapter 3 explains how a KeY proof obligation is translated to SMT. Chapter 4 shows the most notable aspects of the implementation. Chapter 5 provides the experimental results for running our tool on valid and invalid proof obligations. Finally, Chapter 6 provides a summary and presents related and future work.

2. Preliminaries

2.1. JavaDL and KeY

In order to prove a certain property regarding a program, we need a logical framework that allows the formulation of this property. Dynamic logics [HKT84] are logics used to reason about properties of programs. Beckert [Bec01] introduced *JavaDL*, a dynamic logic for a subset of the Java programming language, called Java CARD. This logic allows us to express properties such as termination, non-termination, or fulfillment of user specified method contracts and class invariants for Java programs. The version of JavaDL presented in this section is the one introduced by Weiß [Wei11] which was extended to provide support for heaps, and serves as the de facto specification of the dynamic logic used by KeY.

2.1.1. The Type System

JavaDL has a hierarchical type system, shown in Figure 2.1, which contains the type Any as a supertype for all other types, except $Heap$ and $Field$. The default semantics of the $Integer$ type is that of the mathematical integers. Java integers, with or without overflow checking represent the other two possible semantics, which the user can choose. The $Object$ type is the equivalent to the $java.lang.Object$ Java type, and all Java reference types defined by the user are contained as subtypes of $Object$. The Java type hierarchy is preserved. The $Sequence$ type is used for modelling lists and the $Heap$ and $Field$ types are used for modelling the Java heap memory. The type $LocSet$ represents a set of locations on the heap. A location is a pair $(Object, Field)$.

It is important to note that the type hierarchy is not considered to be final. The rules of the sequent calculus do not depend on the number of subtypes of a certain type. For example, if there are no other Java reference types declared besides $java.lang.Object$, we may not conclude that there are no objects of other, not yet known, reference types. This way additions to the type hierarchy do not affect the correctness of previous proofs. This property is called *modularity*.

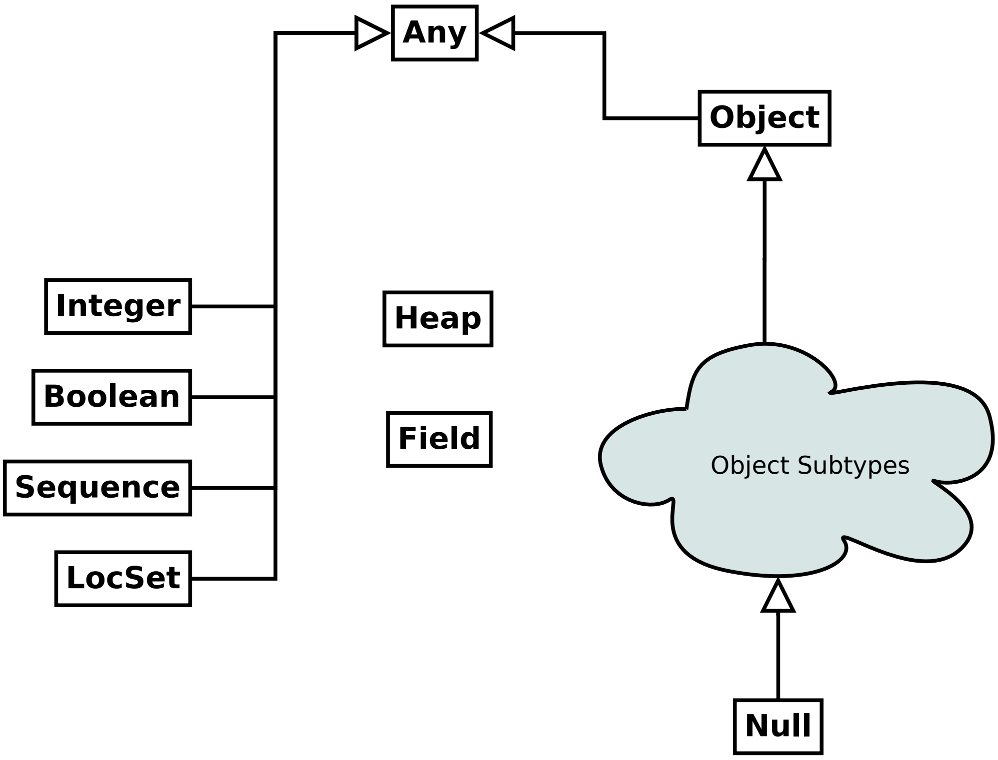

Figure 2.1.: The JavaDL Type System

2.1.2. Syntax

JavaDL is a multimodal extension of a typed first order logic. In addition to the variables, functions and predicates of first order logic, JavaDL provides the box $[\pi]$, diamond $\langle\pi\rangle$ and update U operators.

Definition 2.1 (JavaDL Signature)**.** *A JavaDL signature is a tuple:* $(\tau, \preceq, V, PV, F, F_u, P, \alpha, Prg)$ *where:*

- τ *is the set of types*
- $\preceq$ *is a partial order on* τ*, the subtype relation*
- V *is the set of logical variable symbols*
- PV *is the set of program variable symbols*
- F *is the set of functions symbols*
- $F_u \subseteq F$ *is the set of unique function symbols*
- P *is the set of predicates symbols*
- α *is a static typing function providing a type signature for every symbol:* $\alpha(v) \in \tau$ *for all* $v \in V \cup PV$*, and* $\alpha(f) \in \tau^* \times \tau$ *for all* $f \in F$*, and* $\alpha(p) \in \tau^*$ *for all* $p \in P$
- Prg *is a Java Card program*

Definition 2.2 (Terms)**.** *Given a JavaDL signature* $(\tau, \preceq, V, PV, F, F_u, P, \alpha, Prg)$*, we define the set* $Term_A$ *of terms of type* A *as follows:*

- $x \in Term_A$ *for all* $x \in V \cup PV$ *such that* $\alpha(x) = A$
- $f(t_1, t_2, \dots t_n) \in Term_A$ *for all* $f \in F$, *and* $\alpha(f) = (B_1, B_2 \dots, B_n, A)$, *and* $t_1 \in Term_{B'_1} \dots t_n \in Term_{B'_n}$, *and* $B'_1 \preceq B_1 \dots B'_n \preceq B_n$
- $if\ \phi\ then\ t_1\ else\ t_2$ *for all* $t_1, t_2 \in Term_A$ *and* $\phi \in Formula$
- $\{U\}t$ *for all* $t \in Term_A$

The update operator $\{U\}$ is used to model state transitions.

Definition 2.3 (Updates)**.** *Given a JavaDL signature* $(\tau, \preceq, V, PV, F, F_u, P, \alpha, Prg)$, *we define the set Update which contains functional, parallel, and updates applications:*

- $(v := t) \in Update$ *for all* $v \in PV$ *and* $t \in Term_A$ *such that* $\alpha(v) = A$
- $(u_1 \parallel u_2) \in Update$ *for all* u_1 *and* $u_2 \in Update$
- $(\{u_1\}u_2) \in Update$ *for all* u_1 *and* $u_2 \in Update$

The Java program Prg is composed of one or more sub-programs which we shall call program fragments. For a program fragment *pr* we will use the notation $pr \in Prg$ meaning that *pr* is a program fragment of Prg.

Definition 2.4 (Formulae)**.** *Given a JavaDL signature* $(\tau, \preceq, V, PV, F, F_u, P, \alpha, Prg)$ *we define the set Formula of JavaDL formulae as follows:*

- $true, false \in Formula$
- $p(t_1, t_2, \dots t_n) \in Formula$ *for all* $p \in P$ *and* $t_1 \in Term_{B'_1} \dots t_n \in Term_{B'_n}$ *and* $\alpha(p) = (B_1, B_2 \dots, B_n)$ *and* $B'_1 \preceq B_1 \dots B'_n \preceq B_n$
- $\neg\phi, \phi \wedge \psi, \phi \vee \psi, \phi \rightarrow \psi, \phi \leftrightarrow \psi \in Formula$ *for all* $\phi, \psi \in Formula$
- $\exists x; \phi\ \forall x; \phi \in Formula$ *for all* $x \in Var$ *and* $\phi \in Formula$
- $\{u\}\phi \in Formula$ *for all* $u \in Update$ *and* $\phi \in Formula$
- $[\pi]\phi, \langle\pi\rangle\phi \in Formula$ *for all* $\pi \in Prg$ *and* $\phi \in Formula$

The following functions and predicates will be in every JavaDL signature:

- $instance_T(Any) \in P$
- $exactInstance_T(Any) \in P$
- $cast_T : Any \rightarrow T$

We shall refer to JavaDL formulae without modal operators or updates as KeY first order logic (KeYFOL) formulae.

2.1.3. Semantics

Java programs operate on states. A state s can be thought of as the contents of the memory at a certain point in the execution of a Java program.

Given a beginning state and a Java program, running the program will change the state of the system. The end state achieved after running a Java program depends, of course, on the state the system had before the execution of a program.

Definition 2.5 (Kripke Structure)**.** *We define a Kripke structure as a tuple* (D, δ, M, S, ρ) *where*

- D *is a set of semantic values*
- δ *is a dynamic typing function* $\delta : D \to \tau$ *generating subdomains* $D^A = \{x \in D | \delta(x) \preceq A\}$ *for all* $A \in \tau$
- M *is a first order logic model, which assigns to every symbol in* $F \cup P$ *its semantics.*
- S *is a set of states* s *which are functions mapping program variables* v *of type* A *to values in* D^A
- ρ *is a function which associates a transition relation to each program fragment* pr *such that* $\rho(pr) \subseteq S \times S$ *and* $(s_1, s_2) \in \rho(pr)$ *if* pr *starting from state* s_1 *terminates in* s_2 *and no exception is thrown. Because Java Card programs are deterministic, for each starting state, there can be at most one end state.*

The following functions and predicates have the same semantics in all Kripke structures:

- $instance_T(x) = \{x \in D | \delta(x) \preceq T\}$
- $exactInstance_T(x) = \{x \in D | \delta(x) = T\}$
- $cast_T(x) = \begin{cases} x & \text{if } x \in D^T \\ null & \text{if } x \notin D^T \wedge T \preceq Object \\ empty & \text{if } x \notin D^T \wedge T = LocSet \\ false & \text{if } x \notin D^T \wedge T = Boolean \end{cases}$

We will now present how the transition relation is defined for a selection of Java program fragments.

Let us consider a variable assignment without side effects. $(s, s') \in \rho(x := t)$ *iff* the state s' contains all variable assignments from the state s except for x which will now have the value of the term t, evaluated in state s, assigned.

Java programs can be sequentially combined. Let p_1 and p_2 be two Java programs. $(s, s') \in \rho(p_1; p_2)$ *iff* there is a state r such that p_1 beginning in state s will end in state r and p_2 beginning in state r will end in state s'.

For the program fragment pr consisting of the conditional statement $if(\phi)$ *then* p_1 *else* p_2, $(s, s') \in \rho(pr)$ *iff* either ϕ is true in s and s' is the end state of p_1 when starting in s or ϕ is false in s and s' is the end state of p_2 when starting in s.

Finally,for the program fragment *pr* consisting of a loop statement $while(\phi)\ do\ p$ $(s, s') \in \rho(pr)$ *iff* there exists a sequence $s_1, s_2, \ldots, s_n$ of states, such that p starting in state s_i will end in state s_{i+1} and for all but the last state in the sequence the formula ϕ must be true. In the last state it must be false.

Because each program determines a transition relation between states, JavaDL is called a *multi-modal* logic, *modal* logic is similar, but has only one transition relation for the entire system.

JavaDL provides for each program fragment *pr* two modal operators: The box operator $[pr]$ and the diamond operator $\langle pr \rangle$. Formulae containing these operators have the form $[pr]\phi$ or $\langle pr \rangle \phi$. The formula $[pr]\phi$ is true in state s iff for the program fragment *pr* starting in s the formula ϕ holds a state s' where *pr* terminates, and *pr* may also not terminate. The formula $\langle pr \rangle \phi$ is true in state s iff *pr* starting in s will terminate in a state s' and ϕ holds in that state.

Furthermore JavaDL provides the Update operator. Updates describe changes that are to be applied to a state. They are similar to substitutions in purpose, but have the advantage that they do not have to be immediately applied. Instead, they can be accumulated with each transition to a new state, and after the program has ended and has reached in the final state, they can be simplified before being applied to the final state, thus simplifying the necessary substitutions.

2.1.4. Sequent Calculus

JavaDL allows us to express different properties for Java programs. We now need a technique for proving such formulae. One such technique is a sequent calculus for *JavaDL* which we will now present.

A calculus for a logic is a rule system with which the validity of a formula of that logic can be proven. The rules that we apply on the formula are syntactic. Each rule application brings the proof into a new state, where another rule can be applied. The goal is to arrive in s final state, that closes the proof.

The sequent calculus for *JavaDL* operates on proof obligations, called *sequents*, that contain two sets of formulae: the first one is called *antecedent*, the second one *succedent*:

$$\underbrace{\psi_1, \ldots, \psi_n}_{antecedent} \Rightarrow \underbrace{\phi_1, \ldots, \phi_n}_{succedent}$$

The formulae in the antecedent are in conjunction, while the formulae in the succedent are in disjunction, a sequent being thus equivalent to the following formula:

$$\psi_1 \wedge \ldots \wedge \psi_n \rightarrow \phi_1 \vee \ldots \vee \phi_n$$

The starting sequent of a proof will always have an empty antecedent with the formula that must be proven in the succedent. For a *JavaDL* formula ϕ the starting sequent of a proof is:

$$\Rightarrow \phi$$

The sequent calculus proves the validity of a formula by showing that it can be inferred from a set of axioms. The sequent calculus uses syntactic rules for searching the axioms from which the validity of the formula can be proven. There are two kinds of rules. The first kind takes a sequent and generate new sequents. These new sequents represent formulae from which the original sequent can be inferred. Because a rule application can generate more than one sequent, we obtain a proof tree. The second type of rules, called closing rules, do not generate any new sequents and mark the sequent as closed. Applying a closing rule on a sequent means that that sequent can be directly inferred from a set of axioms. When proving the validity of a formula using the sequent calculus, the objective is to close all branches of the proof tree. In this section we will present a selection of the rules, the rest can be found in the official KeY book [BHS07] or in [Wei11]. In our notation the rule is applied on the sequent on the bottom and generates the sequents on top.

An example of a basic rule of sequent calculus is AND-LEFT:

$$\frac{\Gamma, \phi, \psi \Rightarrow \Delta}{\Gamma, \phi \wedge \psi \Rightarrow \Delta}$$

This rule removes a conjunction from the antecedent and adds its two arguments as separate formulae in the antecedent. A similar rule is OR-RIGHT.

The rules are applied until a sequent is reached on which a closing rule can be applied. Closing rules are also called axioms. An example of an axiom is the TRUE-RIGHT rule:

$$\frac{}{\Gamma \Rightarrow true, \Delta}$$

This rule states that if $true$ is present among the formulae in the succedent, then the entire sequent is valid. Because the succedent is a disjunction, one $true$ sub-formula is enough for it to evaluate to true. Because the sequent is in fact an implication, if the succedent is valid, the entire sequent will also be valid. A similar closing rule is FALSE-LEFT.

An additional rule of the sequent calculus, which we need to introduce, is the *PULL-OUT* rule:

$$\frac{\Gamma, T = t, p(\ldots, T, \ldots) \Rightarrow \Delta}{\Gamma, f(\ldots, t, \ldots) \Rightarrow \Delta}$$

This rule replaces a (sub)term t of a formula in the antecedent with a constant T, and adds a new formula to the antecedent stating that $T = t$. Since formulae from the succedent can be moved to the antecedent and negated, this rule can be applied to (sub)terms of formulae in the succedent as well.

Because *JavaDL* may contain programs, special rules are needed for symbolically executing these programs. Symbolic execution takes the execution path for all possible input values into account. This is why a program property is proven for all possible inputs. The sequent calculus for *JavaDL* applies a rule for each statement of the program. The new sequents that are thus obtained will no longer contain the processed statement. In the process the formula is updated with information taken from the discarded statement. At the end of the symbolic execution of the program, the formula will contain an empty modal operator, that can be simply discarded using the SKIP rule:

$$\frac{\Gamma \Rightarrow \{U\}\phi, \Delta}{\Gamma \Rightarrow \{U\}[\,]\phi, \Delta}$$

Assignments without side effects in the program are handled by the ASSIGN rule, which removes the assignment and adds an additional update to the formula:

$$\frac{\Gamma \Rightarrow \{U\}\{v := t\}[\ldots]\phi, \Delta}{\Gamma \Rightarrow \{U\}[v = t; \ldots]\phi, \Delta}$$

Another rule for symbolic execution that is needed for the Java language is the IF rule:

$$\frac{\Gamma \vdash \{U\}\ (\psi \rightarrow\ [\alpha_1\,; \ldots]\ \phi),\ \Delta \qquad \Gamma \vdash \{U\}\ (\neg\psi \rightarrow\ [\alpha_2\,; \ldots]\ \phi),\ \Delta}{\Gamma \vdash \{U\}[if\ (\psi)\ \alpha_1\ else\ \alpha_2; \ldots]\ \phi,\ \Delta}$$

This rule generates two sequents. The first sequent handles the case in which the condition of the if statement evaluates to *true*. The if statement inside the box operator is replaced in this case with the statements from the true case. Similarly in the second generated sequent, the if statement is replaced with the statements from the false case, which are executed if the condition evaluates to false.

Finally the last rule that we present for the symbolic execution is one that handles while loop statements. Loops are processed using loop invariants. Loop invariants are first order logical formulae that are true before the loop is entered and also after each iteration. Loop invariants must be specified by the user. The WHILE rule uses a loop invariant to generate three new sequents:

$$\frac{\begin{array}{c}\Gamma \Rightarrow \{U\}Inv, \Delta \\ \Gamma \Rightarrow \{U\}\{A_1\}(Inv \wedge \psi \rightarrow [\alpha]Inv), \Delta \\ \Gamma \Rightarrow \{U\}\{A_2\}(Inv \wedge \neg\psi \rightarrow [\ldots]\phi), \Delta\end{array}}{\Gamma \vdash \{U\}[while\ (\psi)\ \alpha; \ldots]\ \phi,\ \Delta}$$

These three new proof obligations are:

1. *Invariant initially valid:* It must be shown that the invariant was valid before entering the loop. This fact must be shown by the sequent $\Gamma \Rightarrow \{U\}Inv, \Delta$.

2. *Invariant valid after each iteration:* It must be shown that after each iteration the formula Inv holds. The sequent that tries to prove this is: $\Gamma \Rightarrow \{U\}\{A_1\}(Inv \wedge \psi \rightarrow [\alpha]Inv), \Delta$.

3. *Use case:* If the first two goals are proven, then we can remove the while statement and use invariant formula for describing the initial state of the remaining program. This is the purpose of the following sequent:
$\Gamma \Rightarrow \{U\}\{A_2\}(Inv \wedge \neg\psi \rightarrow [\ldots]Inv), \Delta$.

Note that only the information encoded in the loop invariant can be used for the rest of the proof. Anything else that is not contained in the loop invariant is lost. However, some of this lost information may be needed for the remaining proof. One solution would be to add this context information to the loop invariant. Another solution is to use the anonymising updates A_1 and A_2 instead. The context $\{U\}$ can be used after the application of the WHILE rule, but the variables that are modified by the loop are deleted using these anonymising updates. This way we will be able to use any context information that we are allowed to use in the remaining proof.

After the end of the symbolic execution, the updates gathered during the process are simplified and then applied to the remaining formula. The goal is then to prove that the remaining formula is valid.

2.1.5. Heaps

An instance of the $Heap$ type represents the Java heap memory in a certain point in the execution of the program. Heaps can be viewed as two dimensional arrays, with indexes of type $Object$ and $Field$. The contents of the heap are of type Any. Java programs can allocate memory by creating objects using the *new* operator. Because we consider the domain for a type in a Kripke structure to be constant, the creation of new objects is modelled using a special field, *created*, which points to a boolean value inside a heap. An object o is considered created in a heap h if the location $(h, created)$ points to $true$. The *created* field is special, because once set to $true$ for an object in a heap, it can never be set to false again for that object in that heap.

The function $select : (Heap, Object, Field) \to Any$ returns the value stored in the location determined by the object and field arguments in the heap argument. In JavaDL we can access any combination of object and field, even though in Java we can access only fields from the class declaration or fields inherited from supertypes. We call a function which provides information regarding the stored values of instances of the container types $Heap$, *LocSet* and *Sequence observer functions*.

In order to change the stored value of a location in a heap we have to use the $store : (Heap, Object, Field, Any) \to Heap$ function. This function returns a new heap with the new value stored in the location determined by the object and field parameters, and with all other values the same as in the parameter heap. However, the *store* function cannot be used in order to change the *created* field. In order to mark an object as created, the function $create : (Heap, Object) \to Heap$ is used, which creates a new heap in which the given object is created, and all other values remain unchanged. It is impossible to mark a created object as not-created.

Another function, which operates on heaps is the $anon : (Heap, LocSet, Heap) \to Heap$ function, which sets locations of an original heap to anonymous values, but does not change the other locations. This is useful when dealing with loops and method calls, which may change the locations from a specified location set. The first heap argument represents the original heap, the second location set argument represents the locations which may change, and the third heap argument represents the heap from which the anonymous values are taken. The heap $h = anon(h_1, ls, h_2)$ contains the values of h_2 in all locations $l \in ls$ and the values of h_1 in all other locations. Like the *store* function, the anon function does not allow to change the value of locations determined by the *created* field from $true$ to $false$. Additionally values of locations of not created objects are also taken from from h_2.

Arrays are modeled using array fields generated by the injective function arr : $Integer \to Field$. We access field i of an array a in a heap h by calling $select(h, a, arr(i))$. The length of an array is modeled using the $length : Object \to Integer$ function, which provides a length for each object, even for those which are not arrays. Using a function instead of a field makes the length of an object independent from the heap. The creation of an array can be viewed as choosing and array of the desired length and creating it.

The sequent calculus contains rules describing the results of providing the heap functions other than *select* as arguments to the *select* function. Examples of such rules are $selectOfStore$, $selectOfCreate$ and $selectOfAnon$. These three rules substitute certain terms with other terms in the sequent, leaving the rest of the sequent intact, and in order to keep the notation simple, we present only the term substitution for these rules:

$$
\begin{aligned}
select(store(h, o, f, v)o_1, f_1) &\leadsto if(o = o_1 \land f = f_1 \land f \neq created) \\
&\quad\; then\ x \\
&\quad\; else\ select(h, o_1, f_1) \\
select(anon(h_1, ls, h_2)o, f) &\leadsto if(elementOf(o, f, ls) \land f \neq created \land \\
&\quad\; elementOf(o, f, unusedLocs(h))) \\
&\quad\; then\ select(h_2, o, f) \\
&\quad\; else\ select(h_1, o, f) \\
select(create(h, o_1)o_2, f) &\leadsto if(o_1 = o_2 \land o_1 \neq null \land f = created) \\
&\quad\; then\ true \\
&\quad\; else\ select(h, o_2, f)
\end{aligned}
$$

Additionally, there is a predicate $wellformed(Heap)$, which states for a heap h that:

1. All objects referenced in h are created in h or equal to $null$
2. All location sets inside h contain locations of object which are created or equal to null
3. The heap contains finitely many created objects
4. The values stored in the heap are of the correct type, i.e. the type from the Java class declaration

For specifying and reasoning about properties of a set of locations on a heap, the *LocSet* type is used. The observer function for location sets is the $elementOf$: $Object \times Field \times LocSet$ predicate, which is true iff the given location is in the location set. The standard set operations are also defined for the location sets.

Similarly to heaps, which were defined using the select function, the semantics of the location set functions is defined using the $elementOf$ observer function. Location sets support set specific functions and predicates like $singleton : Object \times Field \to$

LocSet, $union : LocSet \times LocSet \to LocSet$ and $subset : LocSet \times LocSet \to Bool$ specified by the rules elementOfSingleton, elementOfUnion and subsetToElementOf respectively:

$$\begin{aligned}
elementOf(o, f, singleton(o_1, f_1)) &\leadsto o = o_1 \land f = f_1 \\
elementOf(o, f, union(l_1, l_2)) &\leadsto elementOf(o, f, l_1) \lor elementOf(o, f, l_2) \\
subset(l_1, l_2)) &\leadsto \forall o : Object\ \forall f : Field\ elementOf(o, f, l_1) \\
&\quad \to elementOf(o, f, l_2)
\end{aligned}$$

In order to reason about list data structures more easily, the *Sequence* type is used. Sequences are one dimensional arrays with an index of type *Integer* and contents of type *Any*. Sequences have two observer functions: the length function $seqLength : Sequence \to Integer$ and the get function $seqGet : Integer \to Any$. The supported operations include defining new sequences using the $seqDef$ function, concatenating two sequences using the *concat* operation and getting the subsequence using the *sub* function.

The sequent calculus rules for sequences define the semantics of the operation by using the *seqGet* and *seqLength* functions.

For heaps, location sets and sequences *extensionality rules* define the equality of two terms of one of these types by using the observer functions. The extensionality rules are equalityToSelect, equalityToElementOf and equalityToSeqGetAnSeqLength:

$$\begin{aligned}
h_1 = h_2 &\leadsto \forall o : Object\ \forall f : Field\ select(h_1, o, f) = select(h_2, o, f) \\
l_1 = l_2 &\leadsto \forall o : Object\ \forall f : Field\ elementOf(o, f, l_1) \leftrightarrow elementOf(o, f, l_2) \\
s_1 = s_2 &\leadsto seqLength(s_1) = seqLength(s_2)\ \forall i : Int\ seqGet(s_1, i) = seqGet(s_2, i)
\end{aligned}$$

2.1.6. KeY

KeY is a deductive software verification system for programs written in Java. For this purpose it uses JavaDL and a sequent calculus for JavaDL.

Given a starting sequent, KeY will try to automatically apply the rule it considers most suited. It can do that in accordance with different strategies for applying rules. These rule application strategies are implemented as so-called macros.

In many cases KeY is able to close all proof goals on its own. For more complex problems user input is needed in order to continue the proof. The user may be required to provide an instantiation for a universally quantified formula or to specify an induction rule, among other things.

KeY can generate the initial sequent from the Java source code specified with JML. The rules of the sequent calculus are written as so-called *taclets* using a special language.

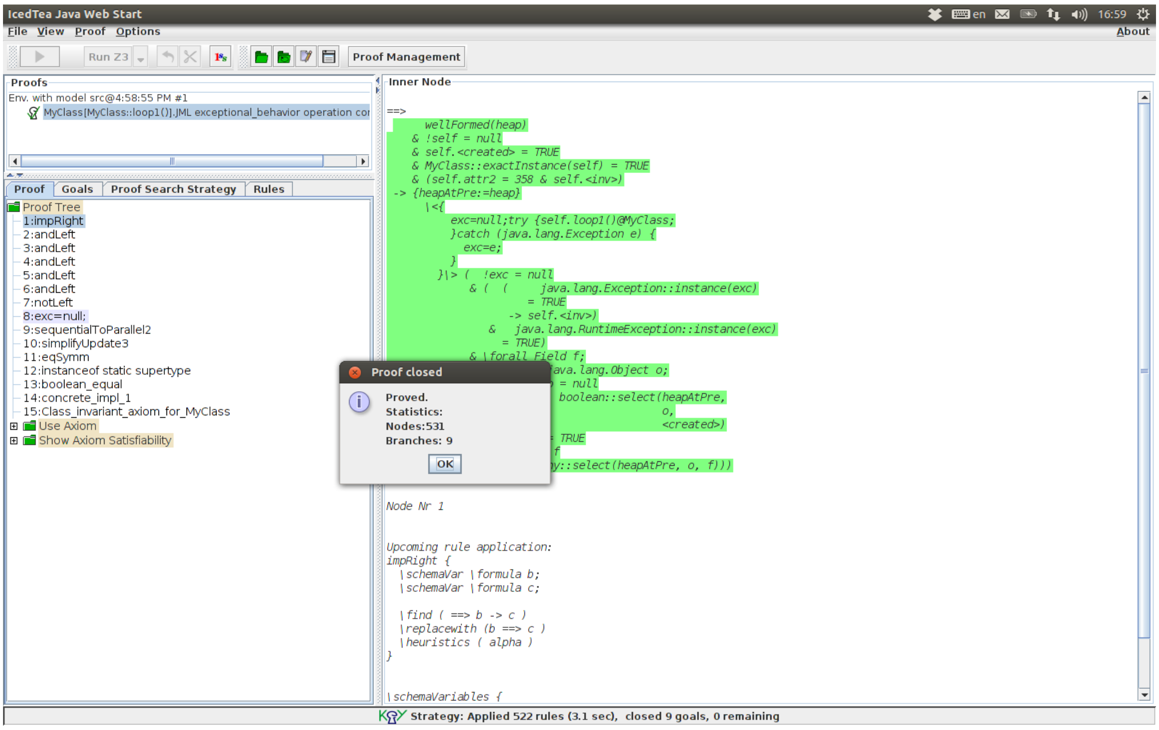

Figure 2.2.: KeY

2.2. SMT

Satisfiability modulo theories (SMT) solvers check the satisfiability of a set of first order logic formulae. As opposed to automatic theorem provers (ATP), which only support pure first order logic formulae, SMT-solvers provide support for background theories. These background theories provide the interpretation for sorts, functions and predicates, which can be used in the first order formulae given to the solver.

The user is not required to provide axioms for the background theories, the background theory does not even need to be first order axiomatizable, and SMT solvers can use dedicated solvers for the supported theories. SMT solvers are fully automatic. If the satisfiability of the formulae is proven, the solver can provide a model which satisfies them.

2.2.1. The SMT-LIB 2 Language

The SMT-LIB 2 language is a standardized [BST12] specification language supported by most SMT solvers. It is used for writing the formulae whose satisfiability needs to be proven. We shall call the set of formulae written in the SMT-LIB language an SMT specification and the SMT-LIB language, simply, SMT.

2.2.2. SMT-LIB commands

An SMT specification is a sequence of commands. We shall present the commands which are used by our tool.

The *declare-sort* command is used to declare new sort symbols. Because SMT sorts can be parametrized using other Sort symbols, the declare-sort function requires the arity of the sort, besides the name of the sort. Since we do not use parametrized sorts, all sorts in the specifications we generate have arity 0. For example the declaration of an SMT sort *Heap* would look like this: *(declare-sort Heap 0)*. All declared sorts have disjoint domains, and no built in interpreted subtype relation is provided. SMT sorts declared like this are uninterpreted, meaning that the solver can come up with any domain for it, as long as this domain is disjoint from the domains of all other declared sorts. Sorts declared using this command have a domain of infinite size and are called unbounded sorts. Not all sorts are declared by the users, there are sorts provided by the background theories like Bool, Int and BitVec. The Bool and BitVec sorts, representing boolean and bit-vector values have finite domains.

The *define-sort* command specifies an additional name for an existing sort. *(declare-sort Heap (_ BitVec 3))* assigns the bit-vector sort of bit-size 3 to the symbol *Heap*. We say *Heap* is an alias of *(_ BitVec 3)*. From the perspective of the solver, two aliases of the same sort are treated as equal sorts. This can lead to unexpected effects. If we define a sort *Object* as the bit-vector sort of size 3 we could provide instances of the Object sort to functions expecting instances of the Heap sort.

The *declare-fun* command declares a function signature. For example *(declare-fun select (Heap Object Field) Any)* declares a function with the name *select* which expects a Heap, Object and Field parameter and return an instance of the Any sort. In this example, the sorts Heap, Object, Field, and Any are declared or defined by the user. Declared functions are uninterpreted, because they do not have any semantics. The SMT solver is free to chose any semantics for a declared function, as long as the chosen semantics satisfies the SMT specification.

The *define-fun* command defines a function. This command takes the function name, a list of parameter names and sorts,the sort of the returned value and a term representing the method body. For example *(declare-fun addOne ((x Int)) Int (+ x 1))* defines a function *addOne* which return an Int equal to the incremented value of the parameter x of type Int. Recursive functions cannot be defined using this function.

The *assert* command adds a formula to the specification. It has the form *(assert F)* where F is an SMT formula. We shall call a formula added this way to the specification an *assertion*.

The *check-sat* command initiates the satisfiability check of the specification composed by the previous commands. This command has no parameters.

The *get-model* command asks the SMT solver to provide a model in the case in which the specification is satisfiable. A model provides a definition to all sorts and functions that were only declared in the specification.

For specifications for which a model was found, terms can be evaluated using the *get-value* command. The evaluation is done using the function definitions from the model. The command takes a term as an argument.

2.2.3. SMT Formulae

In this section we will present a subset of the SMT-Lib language, which is used by our tool. A complete reference can be found in the SMT-Lib standard [BST12].

Definition 2.6 (SMT Term)**.** *We define the set $Term_{SMT}$ of SMT Term as follows:*

- $v \in Term_{SMT}$ *for all variables v quantified by an enveloping quantifier*
- *(f t_1 t_2 ... t_n) in*$Term_{SMT}$ *for all SMT functions f and SMT term t_1 to t_n such that the sorts of the terms t_1 to t_n correspond to the sorts of the expected parameters of f.*
- *(ite cond t_1 t_2) such that cond* $\in Formula_{SMT}$ *and* $t_1, t_2 \in Term_{SMT}$

Definition 2.7 (SMT Formula)**.** *We define the set $Formula_{SMT}$ of SMT formulae as follows:*

- $true, false \in Formula_{SMT}$
- *(and f_1 f_2), (or f_1 f_2), (=> f_1 f_2), (not f_1), (= f_1 f_2)* $\in Formula_{SMT}$ *for all* $f_1, \ldots f_n \in Formula_{SMT}$
- *(p t_1 ... t_n)* $\in Formula_{SMT}$ *for all functions returning Bool and* $t_1 \ldots t_n \in Term_{SMT}$ *such that the sorts of* $t_1 \ldots t_n$ *match the sorts of the parameters of* p
- *(forall ((v S)) f)* $\in Formula_{SMT}$ *for all* $f \in Formula_{SMT}$ *and* $v \in Term_{SMT}$ *such that v has the sort S*
- *(exists ((v S)) f)* $\in Formula_{SMT}$ *for all* $f \in Formula_{SMT}$ *and* $v \in Term_{SMT}$ *such that v has the sort S*

Note that in the SMT-LIB standard [BST12] formulae are considered to be terms of sort Bool. We distinguish between them in order to highlight the similarities between SMT and JavaDL. The functions and, or, =>, not and the quantifiers exists and and forall have the same semantics as in standard first order logic. The = symbol is interpreted as the equivalence function, when used with formulae, and equality otherwise.

2.2.4. Built in sorts and functions

In this section we present the built in functions which we used for our work. The specifications generated by our tool use the built-in sorts Bool and BitVec. All KeY sorts, except boolean are translated as aliases of bit-vectors of different lengths. The SMT bit-vector sort, BitVec is a parametrized sort which takes an integer as an argument, representing the bit-size. For example the bit-vector sort of bit-size 5 is specified like this: (_ BitVec 5).

The background theory of the SMT soler provides the interpreted functions shown in table 2.1.

Instance of the bit-vector sort have the form (_ bv{value} size). For example, (_ bv2 3) is the bit-vector value 2 of size 3: 010.

2.2.5. Z3

Z3 [DMB08] is an SMT solver developed by Microsoft. It is currently one of the best performing SMT solvers for specifications containing uninterpreted functions and quantified bit-vectors, which is the reason why we chose it for our tool.

Function	*Description*
bvadd	Adds two bit-vectors
bvsub	Subtracts two bit-vectors
bvmul	Multiplies two bit-vectors
bvsdiv	Divides two bit-vectors
bvslt	Signed lower than
bvsle	Signed lower than or equals
bvsgt	Signed greater than
bvsge	Signed greater than or equals
concat	Concatenates two bit-vectors
extract	Extracts a sub-range from a bit-vector

Table 2.1.: Built-in functions for bit-vectors

3. Translation

In order to search for a counterexample of a KeY proof obligation we translate the negation of that proof obligation to SMT, and provide the translation to the solver. Because SMT solvers only support first order logic, we can only support proof obligations which do not contain modal operators or updates. The KeY system is capable of automatically applying the rules for symbolically executing a program and applying all updates. Before the translation, the proof obligation is preprocessed as described in Sections 3.3.2 and 3.5. Table 3.1 shows an overview of the translation. The function τ translates terms and formulae from KeYFOL, described in Section 2.1.2, to SMT, described in Section 2.2.3. The rest of this section handles the translation of interpreted functions and predicates.

3.1. The Type System

The KeY type system is specified using 8 SMT sorts: Bool, IntB, Heap, Object, Field, LocSet, SeqB, and Any. Except for the built in sort Bool, all SMT sorts are aliases of bit-vector sorts of different lengths. All KeY reference types, are translated to the *Object* sort. The mapping of KeY types to SMT sorts is presented in Table 3.2.

For some SMT sorts the bit size is specified by the user, for others it is calculated automatically. The user can specify the bit size for the *IntB*, *Object*, *LocSet* and *SeqB* sorts. The bit sizes for the $Heap$ and $Field$ sorts are calculated by taking the logarithm of the number of constants of the respective type in the proof obligation. The bit size of the *Any* sort is computed by taking the maximum bit size of all SMT sorts, which are subtypes of *Any*, and adding three bits for type information in order to distinguish between the five subtypes of *Any*.

Depending on the bit sizes chosen by the user, some SMT sorts may end up as aliases of bit-vectors of the same length. This will not cause any errors, since we cast the terms to *Any* when they appear in an equality. This way the equality between the instance of $IntB$ corresponding to the bit-vector 0 and the instance of $Object$ corresponding to the same value will evaluate to $false$ because of the different type bits of the two sorts. Additionally, all proof obligations that need to be translated

KeYFOL	SMT
$\tau(v)$, where v is an integer value	(_ bv$\{v \pmod{2^{intsize}}\}$ intsize)
$\tau(x)$, where x is a JavaDL variable or constant	X, where X is an SMT variable or constant
$\tau(p(t_1, t_2, \ldots t_n))$	(p $\tau(t_1)$ $\tau(t_2)$ $\ldots$ $\tau(t_n)$), also declares a boolean function p of the appropriate types if not already done
$\tau(f(t_1, t_2, \ldots t_n))$	(f $\tau(t_1)$ $\tau(t_2)$ $\ldots$ $\tau(t_n)$), also declares a function f of the appropriate types if not already done
$\tau(if\, \phi\, then\, t_1\, else\, t_2)$	(ite $\tau(\phi)$ $\tau(t_1)$ $\tau(t_2)$)
$\tau(\forall x : T\ \ F)$	(forall ((x $\tau(T)$)) $\tau(F)$)
$\tau(\exists x : T\ \ F)$	(exists ((x $\tau(T)$)) $\tau(F)$)

Table 3.1.: Overview of the translation

KeY Type	SMT Sort
Boolean	Bool
Integer	IntB
Heap	Heap
Object	Object
Field	Field
LocSet	LocSet
Sequence	SeqB
Any	Any

Table 3.2.: Mapping of KeY types to SMT sorts

are correctly typed, meaning that all functions and predicates have terms of the appropriate type as arguments, since this property is required by the the JavaDL syntax itself.

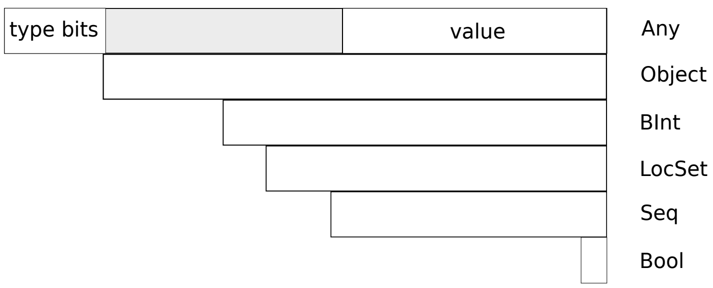

Figure 3.1.: The SMT sorts

For each SMT sort S except Any, Heap, and Field membership predicates and cast functions are declared, which check if an instance of Any is of type S, and cast between S and Any. We declare the following functions for an SMT sort S:

1. $isS : Any \rightarrow Bool$

2. $Any2S : Any \rightarrow S$
3. $S2Any : S \rightarrow Any$

Each SMT Sort except Any, Heap, and Field has a unique bit pattern associated with it, which is used to encode the actual subtype of *Any* as shown in figure 3.1. The membership function simply checks if the appropriate bit pattern is used as type bits. When casting from Any to S we need to find out the type of the instance by looking at the type bits and then extract the bit-vector of the according size from the right part of the instance if the type its match. If the instance of *Any* is of the wrong type, the function returns the *null* and *empty* constants when casting to *Object* and *LocSet* respectively. For the other sorts the result is left unspecified. When casting from an SMT sort S to an Any we concatenate the type bits and, if necessary, some fill up bits to the left.

In order to specify the Java reference types we define the following two predicates for each reference type T:

1. $instance_T : Object \rightarrow Bool$
2. $exactInstance_T : Object \rightarrow Bool$

For an Object o and a reference type T, $instance_T(o)$ is true if o is of type T, and $exactInstance_T(o)$ is true if o is of type T but not of any subtype of T or $null$.

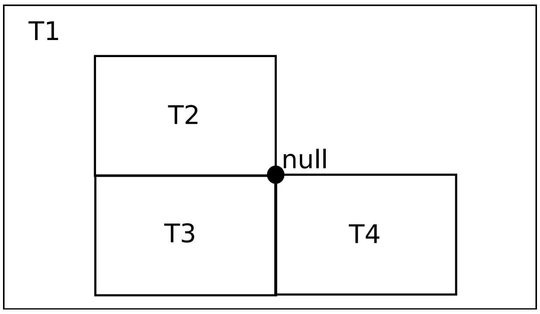

Figure 3.2.: Example types for exactInstance specification

Let T be a reference type and $C1$, and $\ldots Cn$ be the children of T. Then we add the following assertion regarding the $exactInstance_T$ predicate:

$$\forall o : Object \;\; exactInstance_T(o) \rightarrow instance_T(o) \wedge$$
$$\neg(instance_{C1}(o) \vee \ldots \vee instance_{Cn}(o)) \wedge o \neq null$$

The assertion states that if o is an exact instance of T, then it is of type T and not of the type of any child of T and different from $null$. In the example presented in Figure 3.2, $T1$ is the supertype of $T2$, $T3$ and $T4$ while $null$ is the subtype of all other types. In this case an object o being an exact instance of $T1$ implies that it is not type of $T2$, $T3$ or $T4$ and that it is not $null$. The reverse implication is not valid, because it would violate the modularity property of KeY, as explained in Section

2.1.1. Thus, the existence of objects different from *null* of an unknown subtype of $T1$ must be permitted.

We distinguish reference types resulting from interface declarations and reference types resulting from class declarations. The difference between these two categories is that multiple inheritance is allowed for interfaces, but not for classes. Additionally, we distinguish between abstract and concrete reference types. Abstract reference types result from interface and abstract class declarations in Java. There are no objects, which are exact instances of the abstract reference types. Concrete reference types result from concrete class declarations in Java, and allow for exact instances. In order to specify the type hierarchy for the reference types, we add assertions regarding the two predicates for each reference type T. For the different categories of reference types, we need to specify the following:

- *interfaces*: multiple inheritance allowed, exact instances not allowed
- *abstract classes*: multiple inheritance not allowed, exact instances not allowed
- *concrete classes*: multiple inheritance not allowed, exact instances allowed

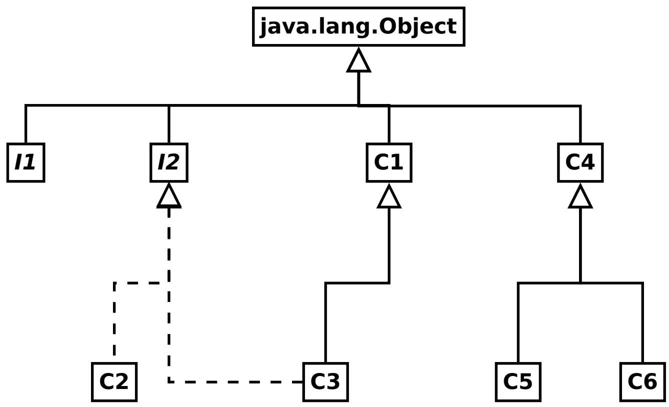

Figure 3.3.: Type hierarchy example. I1 and I2 are interfaces, C1 to C5 are classes.

Let T be a reference type resulting from an interface declaration, $P1, \dots Pn$ the parents of T. We add the following assertion regarding the $instance_T$ predicate:

$$\forall o : Object\ instance_T(o) \rightarrow (instance_{P1}(o) \wedge \dots \wedge instance_{Pn}(o) \wedge \neg exactInstance_T(o)) \vee o = null$$

The assertion states that an object of an interface reference type T is type of all parents of T and not an exact instance of T. For the example presented in figure 3.3, we assert that an object of type $I2$ is also type of its parent, $java.lang.Object$, and not an exact instance of $I2$. By not allowing an object of an interface type to be an exact instance of that type, we force it to be an exact instance of one of its (indirect) subtypes. In order to respect the modularity property of KeY, we must allow the existence of objects of an interface type I, which are not instances of one of the known subtypes of I. In our example we cannot simply state that an object of type $I2$ is either an object of type $C2$ or an object of type $C3$; we allow objects, different from *null*, of type $I2$ which are neither of type $C2$ nor of type $C3$. For the same reason, in the case of the interface type $I1$ we allow for objects different from *null* of that type to exist.

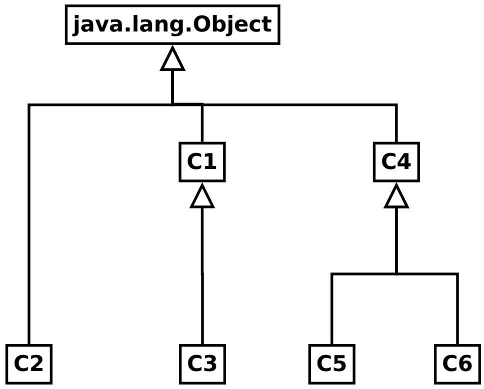

Figure 3.4.: Concrete type hierarchy example

For a reference type hierarchy we define the *concrete type hierarchy* as the type hierarchy which is obtained when contracting all interface types, which means that we remove all interface types from the type hierarchy. All non-interface types with only interface types as parents, will become direct subtypes of the *java.lang.Object* type. The concrete type hierarchy, which results from the type hierarchy shown in figure 3.3 is shown in figure 3.4. The interface types $I1$ and $I2$ have been contracted. The type $C2$ is now a direct subtype of *java.lang.Object*. Since the type $C3$ had a concrete supertype, the contraction has o effect on it, other than the removal of its interface parents.

Let T be a concrete class reference type, $P1, \ldots Pm$ the parents in the original type hierarchy , and $S1, \ldots Sn$ the siblings of T in the concrete type hierarchy. We add the following assertion regarding the $instance_T$ predicate:

$$\begin{aligned}\forall o : Object\ \ instance_T(o) \rightarrow (&instance_{P1}(o) \wedge \ldots \wedge instance_{Pm}(o) \wedge \\ &\neg(instance_{S1}(o) \vee \ldots \vee instance_{Sn}(o))) \vee o = null\end{aligned}$$

The assertion states that a concrete class type T is also the type of all of its parents, including interfaces, but not the type of its siblings with regard to the concrete type hierarchy. While this assertions allow concrete types to have multiple interface types as parents, it allows only for one non-interface parent, thus disallowing multiple inheritance. Assuming there is a concrete type T with two concrete types as parents, because all concrete types are descendants of the *java.lang.Object* type,it is obvious that T must be subtype of two concrete types S_1 and S_2, such that S_1 and S_2 are siblings, which we do not allow.

For example, in the type hierarchy from figure 3.3 we state that objects of type $C5$ cannot be of type $C6$. Because we state that an object of a concrete type is also the type of its parents, and because we add similar assertions to all its concrete parents we ensure that an object of a concrete type is also type of only those concrete types, which lie on the path to $java.lang.Object$. In our example, by adding similar assertions to all concrete types, we ensure that an object of type $C5$ is not of type $C1$, $C2$, $C3$, and $C6$. We cannot add this assertions for interface types, because in their case, multiple inheritance is possible. In the previous example, we cannot state that an object of type $C5$ is not of type $I1$, because a type could exist, which has both $C5$ and $I1$ as its parents. However, we do wish to state that an object of type $C5$ is not of type $C2$. By removing the interface types from the type hierarchy, before adding these assertions we achieve this goal. In figure 3.4 we can observe that $C2$ has become a sibling of $C4$, and thus a $C4$ object cannot be a $C2$ object, and, since all $C5$ objects are $C4$ objects, a $C5$ object cannot be a $C2$ object.

Let T be an abstract class reference type, $P1, \ldots Pm$ the parents in the original type hierarchy , $S1, \ldots Sn$ the siblings of T in the concrete type hierarchy. We add the following assertion regarding the $instance_T$ predicate:

$$\forall o : Object\ \ instance_T(o) \rightarrow (instance_{P1}(o) \wedge \ldots \wedge instance_{Pm}(o) \wedge \\ \neg(instance_{S1}(o) \vee \ldots \vee instance_{Sn}(o)) \wedge \neg exactInstance_T(o)) \vee o = null$$

The assertion states that an abstract class type T is also the type of all of its parents, including interfaces, but not the type of its siblings with regard to the concrete type hierarchy, and there cannot be any objects which are exact instances of T. This assertion disallows multiple inheritance as well as exact instances for abstract classes.

The assertions for the $instance$ and $exactInstance$ predicates are added only for the types which occur in the proof obligation and for their supertypes up until $java.lang.Object$. Ignoring the other types will have no effect on the correctness of the translation. A model for the specification without the ignored types can be transformed into a model for the specification with the ignored types, by adding the missing $instance$ and $exactInstance$ predicates and interpreting them as $false$. Since the assertions we add for the two predicates, when not ignoring them, are implications with the predicates on the left hand side, these additional assertions will be valid. Additionally the predicates may occur in the assertions regarding the $exactInstance$ predicate of other types, but since it appears in a disjunction, it will also have no effect on the constraint. Unsatisfiable specifications without the ignored types will remain unsatisfiable, because adding assertions cannot make them satisfiable.

For concrete class reference types T we need to assert that if an object o is an exact instance of T, then o is not type of any Interface I which is not a supertype of T. Let T be a concrete class reference type and $I1, \ldots In$ the interfaces which are not supertypes of T. We add the following assertion:

$$\forall o : Object\ \ exactInstance_T(o) \rightarrow instance_T(o) \wedge \\ \neg(instance_{I1}(o) \vee \ldots \vee instance_{In}(o))$$

In the example shown in figure 3.3 we need to state for $C3$ objects that they are not $I1$ objects.

Finally, we add an assertion stating that the *null* constant is of every known reference type. Let $T_1, \ldots T_n$ be all known reference types, we assert that:

$$instance_{T_1}(null) \wedge \ldots \wedge instance_{T_n}(null)$$

3.2. Functions

In general, a JavsDL function or predicate $f : (D_1, D_2, \ldots D_n) \to I$ is translated using a declare-function command in SMT. Some functions, however, are translated using built in SMT functions.

When the return type I is a reference type T, other than Object, an assertion is added stating that for all inputs of the appropriate type, the result of the function f is of type T.

3.2.1. Boolean and Integer Functions

For the translation of boolean and integer functions SMT built in functions are used according to the table below:

JavaDL Function	SMT Function
$\neg$	*not*
$\wedge$	*and*
$\vee$	*or*
$\to$	=>
$\leftrightarrow$	=
=	=
+	*bvadd*
$-$	*bvsub*
$*$	*bvmul*
/	*bvsdiv*
$<$	*bvslt*
$\leq$	*bvsle*
$>$	*bvsgt*
$\geq$	*bvsge*

Table 3.3.: Mapping of basic JavaDL operators to built in SMT operators

We interpret the bit-vectors values representing bounded integer as signed with values ranging from $-\dfrac{|IntB|}{2}$ to $\dfrac{|IntB|}{2} - 1$. For this reason we use the signed SMT comparison predicates.

Integer values are translated to bit-vector values. Should value v exceed the maximum or minimum integer value supported by the bound, the values are calculated as (_ bv($v \pmod{2^{intsize}}$) intsize), where intsize is the bit size of the $IntB$ sort. The result is equivalent to adding the bit-vector value 1 v times, if v is positive, and subtracting 1 v times, if v is negative, starting in both cases from 0 and taking overflows into consideration. For example if the bit size of $IntB$ is 3, the value -5 will be translated as $(-5 \pmod 8) = 3$. The result is the same as subtracting 1 from -4, the minimum integer and obtaining 3, the maximum integer. On the other hand, the value 6 will be translated as $(6 \pmod 8) = 6$, and since we use signed bit-vectors it will be interpreted as -2, the result we would obtain when adding 1 6 times, starting from zero.

3.2.2. Cast Functions for Reference Types

We distinguish between two kind of cast functions. The first type performs casts between SMT sorts, and was covered in Section 3.1. The second type of cast functions perform casts between reference types.

For a reference type T we declare the cast SMT function $cast_T : Object \rightarrow Object$ and add the following assertion:

$$\forall o : Object \;\; cast_T(o) = if(instance_T(o)) \;\; then \;\; o \;\; else \;\; null$$

3.2.3. Special Interpreted Constants

The constants *null*, *empty*, *seqEmpty* and *seqOutside* have the following semantics:

- *null* is defined as the Object with bit-vector value 0.
- *empty* is a LocSet constant for which the $elementOf$ predicate always returns $false$.
- *seqEmpty* is a constant of type SeqB for which the *seqLength* always returns 0.
- *seqOutside* is a constant of type Any which is returned when trying to access a position outside the range of a SeqB.

3.2.4. The Wellformed Predicate

The wellformed property for heaps is modelled using the SMT function $wellformed : Heap \rightarrow Bool$, which is defined as a conjunction of four assertions.

The first assertion states that all objects referenced in the heap are either null or created:

$$\begin{aligned} \forall o : Object \;\; \forall f : Field \;\; & Any2Object(select(h, o, f)) = null \lor \\ & Any2Bool(select(h, Any2Object(select(h, o, f)), created)) \end{aligned}$$

Because the *Any2Object* cast function returns a default value of *null*, in case when the argument is not an object, we do not need to consider this cases.

The second assertion states that all location sets stored in the heap contain only objects which are created or null:

$$\begin{aligned}\forall o : \mathit{Object}\ \forall f : \mathit{Field}\ \forall o_1 : \mathit{Object}\ \forall f_1 : \mathit{Field}\\ \mathit{elementOf}(o_1, f_1, \mathit{Any2LocationSet}(\mathit{select}(h, o, f)))\\ \to o_1 = \mathit{null} \vee \mathit{Any2Bool}(\mathit{select}(h, o_1, \mathit{created}))\end{aligned}$$

Similarly to the *Any2Object* function, the *Any2LocSet* function returns default value of *empty* and we do not need to consider the cases in which the result of the *select* function is not of type LocSet.

The third assertion states that all results of the select function are of the correct type. For each field f of SMT sort S and, optionally, the Java reference type T we assert that:

$$\forall o : \mathit{Object}\ \mathit{isS}(\mathit{select}(h, o, f)) \wedge \mathit{instance}_T(\mathit{Any2Object}(\mathit{select}(h, o, f)))$$

Finally, the fourth assertion states that the contents of arrays stored in the Heap are of the correct type. For each arrayType T[] of smt type S(Object, IntB, Bool depending on T) and possible java reference type T we assert that:

$$\begin{aligned}\forall i : \mathit{IntB}\ \forall o : \mathit{Object}\ \mathit{instance}_{T[]}(o) \wedge o \neq \mathit{null} \to \mathit{isS}(\mathit{select}(h, o, arr(i))) \wedge\\ \mathit{instance}_T(\mathit{Any2Object}(\mathit{select}(h, o, arr(i))))\end{aligned}$$

In order to remain satisfiable, we make an exception for the *null* Object. Because *null* is of every type it is of every array type, and, thus its contents would be of all possible types including IntB and Object, which is impossible.

3.3. Preserving the Semantics of Interpreted Functions

We need to preserve the semantics for all interpreted functions (e.g. the *store* function) which appear in the proof obligation, otherwise the SMT solver will make use of incorrect interpretations for such functions in order generate counterexamples. For example if no semantics is specified for the *store* function, the solver could generate a counterexample in which the store function returns the heap it received as an input, which would be incorrect. Such counterexamples would be spurious, and we must avoid them.

This can be achieved in two ways. We could translate the relevant rules to SMT which would specify the semantics of these functions for all possible inputs, but this approach has numerous disadvantages. As an alternative we could specify the semantics of the functions only for inputs which appear in the proof obligation.

3.3.1. Translating Rules

Using existing functionality in KeY we can translate taclets from the taclet language to KeY first order logic. Then we can perform the translation from KeY first order logic to SMT. In order to specify the semantics of the *store* function the translation of the *selectOfStore* rule is needed:

$$\begin{aligned}
&\forall h : Heap\ \forall o : Object\ \forall f : Field \\
&\forall v : Any\ \forall o_1 : Object\ \forall f_1 : Field \\
&select(store(h, o, f, v), o_1, f_1) = \\
if(o = o_1 \wedge f = f_1 \wedge f \neq\ &java.lang.Object :: created) \\
&then\ v\ else\ select(h, o, f)
\end{aligned}$$

Two problems arise from this approach. First, we need to introduce an assertion containing 6 quantifiers, which affects the performance of the SMT solver. Second, in order for this assertion to be satisfiable the size of the heap sort has to be carefully set. It needs to be large enough to support all possible heaps which can result from the *store* function. We can consider the heap sort a two dimensional array of size $|Object| \times |Field|$ which contains values of type Any. The number of heaps $|Heap|$ which we need to support is $|Any|^{|Object|\cdot|Field|}$. This number is huge, even for examples with few objects and fields, and would severely affect the performance of the SMT solver.

For these reasons we cannot use this approach and we are forced to look for alternatives.

3.3.2. Specifying Semantics only for Necessary Inputs

In order to obtain a correct counterexample it is not always necessary to specify the semantics of interpreted functions for all possible inputs. We can provide a specification for those inputs which appear in the proof obligation.

This is achieved by replacing all interpreted function calls with their semantics. We call this approach *semantic blasting*.

The functions dealing with heaps, location sets, and sequences, however, do not have a direct definition. Their semantics is specified using so called observer functions like *select*, *elementOf*, *get*, and *length*.

For functions and predicates, which do not have to occur as argument of an observer function, semantic blasting is straightforward: we apply the necessary rule. Such an example is the replacement the *subset* predicate by using the subsetToElementOf rule, as described in Section 2.1.5.

There are functions and predicates for which we can perform a straightforward replacement only if they appear as an argument of an observer function. For example for the *store* function we can apply the *selectOfStore* rule only if we encounter a term $select(store(h, o, f, v), o_1, f_1)$ where the store function appears as an argument of the select function.

For the cases in which the interpreted function call is not an argument of an observer function we perform the semantic blasting in three steps:

1. We use the *pullout* rule on the term to replace it with a constant and add an equality to the antecedent
2. We use an extensionality rule on the equality added to the antecedent
3. On the right side of the equation the interpreted function call will appear as an argument of the observer function, and we can apply the appropriate rule.

In the following example, the first sequent contains a function f, which has the union of two location sets A and B as its parameter:

1. $\Rightarrow f(union(A, B))$
2. $U = union(A, B) \Rightarrow f(U)$
3. $\forall o : Object\, \forall f : Field\, elementOf(o, f, U) \leftrightarrow elementOf(o, f, union(A, B)) \Rightarrow f(U)$
4. $\forall o : Object\ \forall f : Field\ elementOf(o, f, U) \leftrightarrow elementOf(o, f, A)$ $\vee elementOf(o, f, B) \Rightarrow f(U)$

We wish to replace the sequent with an equivalent sequent which does not contain the *union* function symbol. We can achieve this by applying the elementOfUnion rule, but we can only apply this rule when we have the *union* function call as an argument of the $elementOf$ function call. In the second step, we apply the pullout rule on the *union* term, which replaces it with a constant U and adds an equality in the antecedent, stating that U is equal to $union(A, B)$. In the third step, we apply the equalityToElementOf rule, which represents the extensionality property for sets and get that all locations, which are element of U are also element of $union(A, B)$. On the right side of the equivalence we now have a term on which we can apply the elementOfUnion rule, which we do in the last step. After completing the four steps we have an equivalent proof obligation, which no longer contains the *union* symbol.

After semantically blasting a sequent we obtain an equivalent sequent, which no longer contains function symbols for heaps, location sets and sequences other than observer functions. Additionally the sizes can be lower than when translating taclets and only the semantics of those functions, which are actually used in the proof obligation, is translated. This technique cannot be applied to recursive functions and the loss of the universally quantified axioms can lead to spurious counterexamples as shown in Section 3.7.

3.4. Fields and Arrays

The bit size of the SMT sort Field is automatically calculated by taking the logarithm of the number of Field constants encountered in the proof obligation and adding the resulting bit size to the bit size of IntB. The bit size of IntB needs to be added because of the way in which arrays are modelled. Arrays are instances of the SMT sort Object. Their contents are accessed using array fields. These array fields are produced by the *arr* function, which is declared as $arr : IntB \rightarrow Field$.

The *arr* function is defined as a function, which simply increases the size of a bit-vector value, by concatenating zeroes to the left, leaving the value intact.

For each named field constant encountered in the proof obligation, we define an SMT constant function of an unique value, not in the image of the *arr* function. This way we ensure that all fields are distinct.

3.5. Class Invariants and Model Fields

The class invariant is modelled using the SMT predicate $classInvariant : Heap \times Object \to Bool$. In order to avoid spurious counterexamples we need to translate the semantics for this predicate. The semantics of this predicate results from the invariant formula specified for each reference type T. Let $inv_T[h, o]$ be the invariant formula for the reference type T. We add the following formula to the antecedent of the proof obligation for each reference T:

$$\forall h : Heap\ \forall o : Object\ \ instance_T(o) \to (classInvariant(h, o) \to inv_T[h, o])$$

The formula states that for objects of reference type T the invariant implies the invariant formula of T. The invariant formula $inv_T[h, o]$ cannot imply the invariant predicate in this case because o may be a subtype of T and its invariant formula may be more specific, as shown in the following example:

Listing 3.1: Invariant Example

```
class C{
        /*@ invariant x >= 5; */
        protected int x;
}

class U extends C{
        /*@ invariant x == 5; */
}
```

Listing 3.1 shows two Java classes C and U which both have an invariant specified in the JML specification language. Suppose we have an object o of type C with $o.x = 6$. The object o satisfies the invariant term of C, because $o.x$ is larger or equal to 5, however we cannot affirm that the invariant for o holds, since o may be of the subtype U, and in this case the invariant would require $o.x$ to equal 5. Even if a class like U is not provided by the user, we still cannot use the reverse implication, because it would violate the modularity property of KeY.

For the case in which it is known that an object o is an exact instance of a reference type T we may replace the second implication with an equivalence and add the following formula to the antecedent:

$$\forall h : Heap\ \forall o : Object\ exactInstance_T(o) \to (classInvariant(h, o) \leftrightarrow inv_T[h, o])$$

Similarly to the class invariants, each reference type T can have several specified model fields, which have a definition term $modelfield[h, o]$. The model fields are translated as functions $modelfield : Heap \times Object \to Any$. In order to preserve the semantics of these functions we add the following formula to the antecedent of the proof obligation:

$$\forall h : Heap\ \forall o : Object\ \ exactInstance_T(o) \to (modelfield(h, o) = modelfield[h, o])$$

The formulae for class invariants and represents clauses are added to the antecedent of the proof obligation in order to be able to semantically blast them, because they may contain interpreted functions.

3.6. Preventing Integer Overflows

When dealing with integer constraints, the solver may find counterexamples using integer overflows. Such a counterexample is spurious when the default KeY integer semantics of mathematical integers is used. For this reason it is necessary to provide additional assertions in order to prevent the solver from finding such a counterexample.

Let us consider the formula $\neg(a > 0 \wedge a+1 > 0)$ where a is an integer constant. The SMT solver will try to find a value larger than zero for a such that $a+1$ will not be larger than zero. This formula is unsatisfiable in the default KeY integer semantics with an infinite number of integers. However, we translate the KeY integer sort as the $IntB$ SMT sort which is an alias for a bit-vector sort. This is why the SMT solver will be able to find a model for this formula: it will assign the largest positive integer value (within the size of $IntB$) $maxInt$ to a and the result of $a+1$ will be $minInt$. This model is spurious.

The general idea is to find all terms which can cause an overflow and increase the bit size of these subterms and assert that the result of the same arithmetic operation on the increased bit-vectors is not greater than $maxInt$ or smaller than $minInt$. We increase the bit-vectors using a function *incr*.

For addition and subtraction we increase the bit-vector size by 1, for multiplication we double the bit-vector size. The size of the bit-vectors increased by using the *concat* function: for positive bit-vectors we concatenate zeroes to the left, for negative bit-vectors we concatenate ones to the left.

For an arithmetic operation *op* which may overflow, and for each term of the form $op(x, y)$ occurring in the proof obligation we generate a guard stating that the result of applying the operation on larger bit-vectors is lower than or equal to the maximum integer:

$$op(incr(x), incr(y)) \leq incr(maxInt)$$

Additionally we generate a guard stating that the the operation on larger bit-vectors is larger than or equal to the minimum integer:

$$op(incr(x), incr(y)) \geq incr(maxInt)$$

These guards are added to the formula of the outer most quantifier such that all quantified variables remain quantified and guards for ground terms are added as separate assertions. For universal quantifiers the guards imply the formula, for existential quantifiers we use conjunction. The supported operations are addition, subtraction and multiplication.

3.7. Limitations of our approach

3.7.1. Spurious counterexamples

The tool currently supports function symbols for the Boolean, Integer (partially), Heap, Field, Object, LocSet, and Sequence (partially) types. Examples for not supported function symbols are *bsum*, *bprod*, and $indexOf$. Should the proof obligation contain a not supported function symbol, it will translate the respective function as an uninterpreted function, giving the SMT solver the liberty of choosing its semantics, which can result in spurious counterexamples.

Even for proof obligations containing only supported functions, we can still obtain spurious counterexamples. The first reason for obtaining a spurious counterexample is the presence of integer values larger than the bound for the type integer. In this case the translation applies the modulo function on those values and the resulting value may cause a spurious counterexample. Assuming the bit-size for the IntB SMT sort is 3 let us consider the following unsatisfiable formula:

$$a = 2 \wedge b = 8 \rightarrow a > b$$

The maximum positive value for the IntB sort is 3. Because the formula contains the value $8 > 3$, the value will be interpreted as $8 \pmod 8 = 0$. The formula that is actually given to the solver is valid:

$$a = 2 \wedge b = 0 \rightarrow a > b$$

For the same reason, the solver may claim a formula is unsatisfiable, when in fact it is not. In order to avoid this kind of problems, the user should set the integer bit-size high enough.

Another reason for obtaining spurious counterexamples is the translation of infinite types as finite types. While in KeY the integer type is unbounded, we translate it to the $IntB$ SMT sort, which is bounded. In such cases, the implicit assertion stating that the KeY type is infinite is lost in the translation. Let us consider the following formula:

$$\forall i : Int \ \exists j : Int \ i < j$$

This formula is obviously valid when using mathematical integers. However, when using a bounded type for integers, there will always be a maximal value, no matter what bit-size is used. This kind of spurious counterexamples cannot be avoided, because they do not depend on the sizes of the SMT sorts.

A third cause for spurious counterexamples is the usage of semantic blasting instead of translating the necessary rules for preserving the semantics of functions and predicates. One of the reasons for using semantic blasting was the fact that many of these rules required the existence of a great number of instances of the Heap, LocSet and Sequence sorts, thus requiring large bit-sizes for these sorts. The problem arises when the existence of an instance of one of these sorts is required in order for the specification to be satisfiable, and if this instance does not appear in the proof obligation. For instance if the specification states that there is a sequence with length 1, the SMt solver may return a spurious counterexample, in which all sequences are of length different than 1, because the semantics of functions needed to construct such a sequence translated entirely.

3.7.2. Increasing confidence in proof obligations

Because all SMT sorts are bounded, the translation cannot be used for proving formulae. If a specification turns out to be unsatisfiable for some SMT sort bounds, we cannot conclude that it is unsatisfiable for all bounds. However, the fact that no counterexample was found may increase the confidence of the user in the validity of the proof obligation.

3.7.3. Deviations from the Current Implementation of KeY

The greatest deviation from KeY is the fact that we use bounded sorts, whereas in KeY all types except Bool are unbounded.

In order for the semantic blasting procedure to work on heaps we had to introduce an extensionality rule for heaps.

The default values *null* and *empty* for casting to *Object* and *LocSet* (the functions $Any2Object$ and $Any2LocSet$) are also deviations from the implementation of KeY, which does not specify any value for the case in which the cast fails.

4. Implementation

4.1. Overview

Before verifying a proof obligation, the user can adjust the following settings:

- timeout: Specify for how long the SMT solver will search for a conclusion
- bit-sizes for the SMT-sorts.

Starting with a proof obligation in KeYFOL the user has to perform the following steps:

1. Use the *Add Class Axioms* macro on the proof obligation for adding the assertions described in Section 3.5.
2. Use the *Semantic Blasting* macro on the proof obligation for semantically blasting the supported non-observer functions as described in Section 3.3.2.
3. Use an SMT solver to perform bounded verification.

There are three possible outcomes to the bounded verification.

1. *valid*: The resulting specification is not satisfiable for the chosen bounds.
2. *timeout*: The solver could not reach a conclusion in the given time.
3. *counterexample*: The solver was able to find a counterexample, the user can analyse the counterexample.

4.2. Semantic Blasting

This section describes the implementation of the semantic blasting procedure presented in section 3.3.2.

Semantic blasting is implemented using a KeY macro. The macro controls the application of three types of rules:

1. Semantics rules
2. Extensionality rules
3. Pullout rules

Semantic rules replace a JavaDL formula containing a function symbol f with an equivalent formula which no longer contains the function symbol. Examples for such rules are the $selectOfStore$, $selectOfCreate$ and $elementOfUnion$ rules.

Extensionality rules replace the equality with the observer functions for the $Heap$, $LocSet$ and $Sequence$ data types in KeY. The extensionality rules are $equalityToSelect$, $equalityToElementOf$ and $equalityToSeqGetAndSeqLength$.

Pullout rules are apply the pullout rule on certain terms. The only terms which are allowed to be pulled out, are those having a functions symbol for which we can apply a semantics rule.

The macro assigns the highest priority to semantics rules, lower priority to extensionality rules and lowest priority to pullout rules.

4.3. Counterexample Extraction

If the translation of the negated proof obligation is satisfiable, the SMT solver will also provide a model serving as counterexample for the proof obligation. In the case of the Z3 solver, the counterexample consists of function definitions. However, these function definitions are very hard for a human to read, because of the large number of auxiliary functions, the solver uses in the function definitions. If we inline the auxiliary definitions, we often get very large nested if-then-else statements, which are also very difficult to read. We address this issue by extracting the values which interest the user and present them in a human readable format.

If a counterexample has been successfully generated by the solver, we extract the relevant values from it and put them in our own model data structure. This data structure can be presented to the user in a various ways.

For the non-auxiliary functions, not all values are of interest to the user. For example, the user will care about the values of the *select* function only for the heaps which appear as constants in the proof obligation, and not for all heap values.

Listing 4.1: Specified Java Class

```java
public class A {
	private int x;
	/*@
	  requires x == 2;
	  ensures x == 4;
	  @*/
	public void f(){
		x++;
	}
}
```

In the Java program specified with JML shown in Listing 4.1 the class A contains a field x of type *int* and a method f which increments the value of x. The method contract of f states that if x is 2 in the initial state, it will be 4 in the post state. This contract can obviously not be fulfilled, and we expect a counterexample in which x is 2 in the initial state and 3 in the post state. The z3 SMT solver does find a counterexample for this example, however the output of the z3 solver, shown in Listing 4.2 is difficult for humans to read. The *select_* function which is of interest in this example is defined using the auxiliary functions *select_*!38, k!35 and k!34 which do not appear in the proof obligation, and are meaningless to the user. In order to make the counterexample more readable for the user, we need to extract the information which is of interest to the user and present it in a user friendly way.

Listing 4.2: z3 output for 4.1

```
(model
  (define-fun empty () (_ BitVec 1)
    #b1)
  (define-fun store_0 () (_ BitVec 1)
    #b1)
  (define-fun elem!28 () (_ BitVec 4)
    #x8)
  (define-fun seqGetOutside () (_ BitVec 6)
    #b101010)
  (define-fun seqEmpty () (_ BitVec 1)
    #b0)
  (define-fun heap () (_ BitVec 1)
    #b0)
  (define-fun self () (_ BitVec 1)
    #b1)
  (define-fun length ((x!1 (_ BitVec 1))) (_ BitVec 3)
    #b000)
  (define-fun seqGet ((x!1 (_ BitVec 1)) (x!2 (_ BitVec 3))) (_ BitVec 6)
    #b101010)
  (define-fun k!35 ((x!1 (_ BitVec 1))) (_ BitVec 1)
    (ite (= x!1 #b1) #b1
      #b0))
  (define-fun k!29 ((x!1 (_ BitVec 1))) (_ BitVec 1)
    (ite (= x!1 #b1) #b1
      #b0))
  (define-fun k!33 ((x!1 (_ BitVec 6))) (_ BitVec 6)
    (ite (= x!1 #b000001) #b000001
      #b000000))
  (define-fun seqLen ((x!1 (_ BitVec 1))) (_ BitVec 3)
    #b000)
  (define-fun Any2IntB ((x!1 (_ BitVec 6))) (_ BitVec 3)
    (ite (= x!1 #b101010) #b010
    (ite (= x!1 #b101000) #b000
      ((_ extract 2 0) x!1))))
  (define-fun exactInstanceOf_A!36 ((x!1 (_ BitVec 1))) Bool
    (ite (= x!1 #b0) false
      true))
  (define-fun exactInstanceOf_A ((x!1 (_ BitVec 1))) Bool
```

```
    (exactInstanceOf_A!36 (k!29 x!1)))
  (define-fun Any2Bool!37 ((x!1 (_ BitVec 6))) Bool
    (ite (= x!1 #b000000) false
      true))
  (define-fun Any2Bool ((x!1 (_ BitVec 6))) Bool
    (Any2Bool!37 (k!33 x!1)))
  (define-fun classInvariant ((x!1 (_ BitVec 1)) (x!2 (_ BitVec 1))) Bool
    true)
  (define-fun k!34 ((x!1 (_ BitVec 4))) (_ BitVec 4)
    (ite (= x!1 #x8) #x8
      #x9))
  (define-fun select_!38 ((x!1 (_ BitVec 1))
   (x!2 (_ BitVec 1))
   (x!3 (_ BitVec 4))) (_ BitVec 6)
    (ite (and (= x!1 #b0) (= x!2 #b1) (= x!3 #x8)) #b101010
    (ite (and (= x!1 #b1) (= x!2 #b1) (= x!3 #x8)) #b101011
    (ite (and (= x!1 #b0) (= x!2 #b0) (= x!3 #x9)) #b000000
    (ite (and (= x!1 #b1) (= x!2 #b0) (= x!3 #x8)) #b101000
    (ite (and (= x!1 #b0) (= x!2 #b0) (= x!3 #x8)) #b101000
    (ite (and (= x!1 #b1) (= x!2 #b0) (= x!3 #x9)) #b000000
      #b000001)))))))
  (define-fun select_ ((x!1 (_ BitVec 1)) (x!2 (_ BitVec 1)) (x!3 (_
      BitVec 4))) (_ BitVec
     6)
    (select_!38 x!1 (k!35 x!2) (k!34 x!3)))
  (define-fun typeof_A ((x!1 (_ BitVec 1))) Bool
    true)
  (define-fun elementOf ((x!1 (_ BitVec 1)) (x!2 (_ BitVec 4)) (x!3 (_
      BitVec 1))) Bool
    false)
)
```

We represent the model internally using the *Model* Java class. A model contains constant values, heaps, location sets and sequences. Heaps contain objects, which contain field values. Additionally, the objects contain information regarding their reference type, their length and their status as exact instance.

There are two ways in which we can extract the necessary data from the generated counterexample. First, we can parse the function definitions provided by the SMT solver in the SMT-LIB 2 language and then evaluate them for the values we are interested in. The main disadvantage in this case is the fact that we need to ensure that our implementation of the evaluation function has the same semantics like the one used by the SMT solver. We also need to offer support for all built in SMT operators, which may appear in the function definitions. A second way to extract the required data is to use the $get - value$ command. This command takes a (ground)term as an argument and returns the result of its evaluation. The main advantage is that we can be sure that the value we get is correct, but we need to manage the communication between KeY and Z3 processes, which is more complicated than the first solution. We have opted for the second solution, because we do not need to support the evaluation of all built in SMT functions.

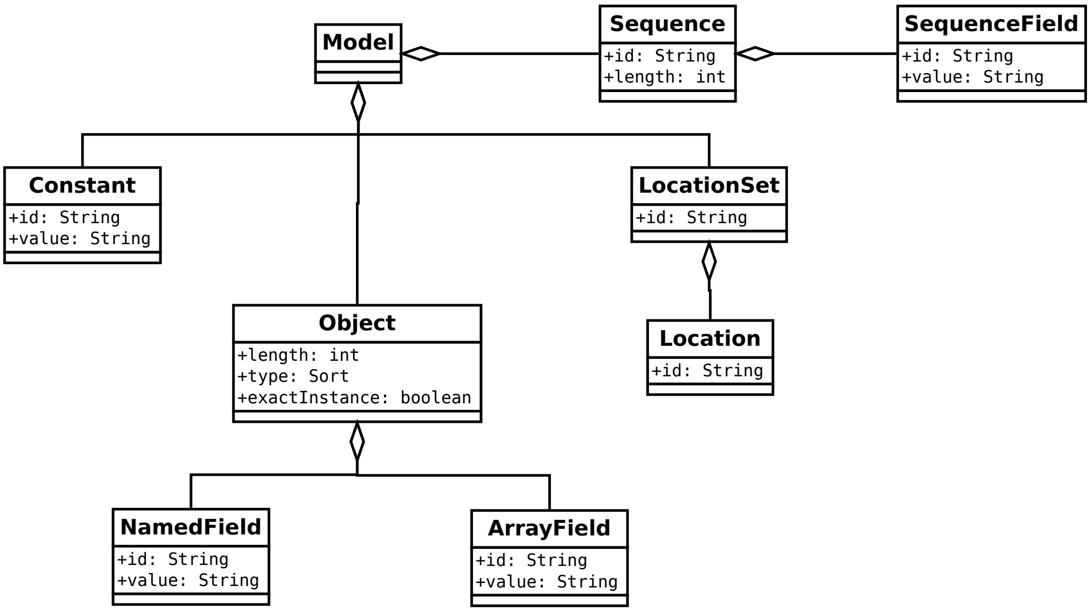

Figure 4.1.: The model data structure

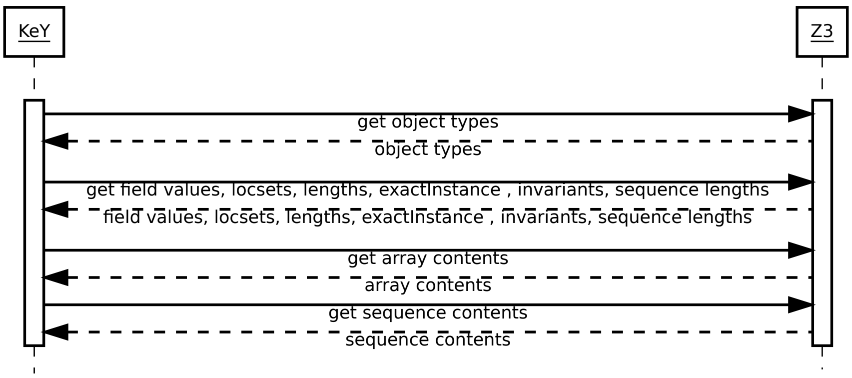

Figure 4.2.: Communication between KeY and the SMT solver

As shown in figure 4.2, for the case in which a counterexample has been found, the data is extracted in the following steps:

1. Extract the type of each object
2. Extract the values for constants, named location sets, and relevant named fields for object, the lengths for all objects, and the lengths for all sequences
3. Extract the values for array fields for objects with length greater than or equal to 1
4. Extract the values for for sequences with length larger than 1

In order to communicate with the solver we use the class *AbstractQuery*, shown in Figure 4.3, which has two methods: $getQuery()$, which returns the $get - value$ command that we wish to send to the solver and the $setResult(String)$ method, which is used to parse and store the response we get from the solver. For each step the necessary Query Objects are created and added to a queue. The queue is then processed and for each element the $get-value$ command is sent to the solver, and the response is then parsed and stored to the element. When all queries have been processed, the parsed responses are then used to add the relevant data to the model.

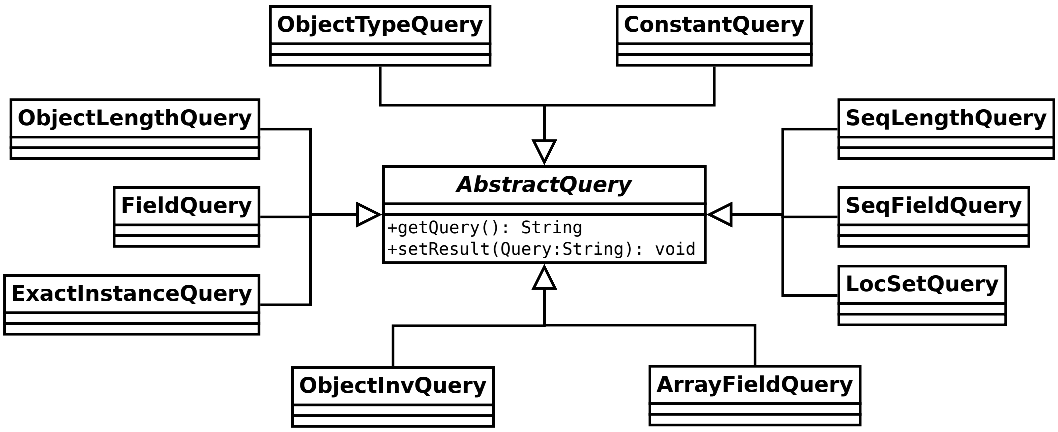

Figure 4.3.: The different query classes

4.4. Counterexample Presentation

The model class can be represented in different ways to the user. The most basic solution, which is currently implemented is to generate a human readable text and show it the the user.

After all the relevant data has been extracted, we format all values v, depending on their type:

- heaps: $\#hv$
- objects: $\#ov$
- fields: $\#fv$
- sequences: $\#sv$
- boolean: $true/false$
- integers: signed decimal format

For example, $\#h1$ represents the heap corresponding to the bit-vector value 1 and $\#o4$ represents the object corresponding to the bit-vector value 5. Additionally, for all object fields, if the value of that fields is equal to a constant than the name of that constant is written next to the value.

For example the text generated by the tool for the example shown in Listing 4.1 we present listing 4.3.

Listing 4.3: A counterexample in text form for 4.1

```
Constants
-----------
heap = #h0
```

```
store_0 = #h1
seqEmpty = #s0
|A::x| = #f8
|java.lang.Object::<created>| = #f9
empty = #l1
seqGetOutside = #a42
self = #o1
null = #o0

Heaps
-----------
Heap heap
   Object #o0/null

   Object #o1/self
      length = 0
      type =A
      exactInstance =true
      |A::x| = 2
      |java.lang.Object::<created>| = true
      classInvariant = true

Heap store_0
   Object #o0/null

   Object #o1/self
      length = 0
      type =A
      exactInstance =true
      |A::x| = 3
      |java.lang.Object::<created>| = true
      classInvariant = true

Location Sets
-----------
#l0 = {}
#l1 = {}

Sequences
-----------
Seq: #s0/seqEmpty
Length: 0

Seq: #s1
Length: 0
```

The text form comprises four sections:

1. Constants
2. Heaps
3. Location Sets

4. Sequences

The Constants section shows the value for each constant.

The Heaps section shows for each heap occurring in the proof obligation the relevant information for all objects in that heap. For each object the length and type are shown. The type shown is the most specific type that could be determined. Additionally for each object we show if it is an exact instance of its type. By determining the type of the object we also find out what fields are declared in its class. The values of those fields are shown next. Furthermore, for each object in each heap we show the result of all functions, which take a heap and an object as arguments. One such function is the *classInvariant* function, and for model fields other such functions may be generated. Finally, for objects with length greater than 0 and of an array type, the values of the array fields are shown.

The Location Sets section displays all location sets with the locations they contain.

The Sequences section displays the length and contents of all sequences.

5. Evaluation

We evaluate the correctness of our application and the feasibility of our approach, by running it on proof obligations which can be automatically closed by KeY, as well as on proof obligations which cannot be automatically closed.

5.1. Proof Obligations Expected to be Valid

Since we consider the sequent calculus and KeY to be correct, our tool should not generate any counterexamples for proof obligations which can be closed by KeY, except for the cases mentioned in section 3.7.

We tested our tool using a bit size of 3 for each sort. The proof obligation we tested are the ones which remain after running the symbolic execution macro. We used a timeout of 5 minutes. The proof obligations presented in Table 5.1 originate from the specification of the Java program B.1. The proof obligations from the table were obtained after symbolically executing the methods and then applying all updates. All methods contracts could be automatically proven by KeY. For the two proof obligations, where we got timeout, we have tried to lower the bit sizes for the integer and object sorts and we got the expected result.

5.2. Proof Obligations Expected to be Invalid

In this section we present the results our tool achieved when processing not automatically closable KeY proof obligations. Besides the bit size of 3, which we also used in the previous section, we also tried increasing it to 4. Again, the timeout was set to 5 minutes.

5.2.1. Specifications with Unknown Faults

In this section we will present the counterexamples for specifications containing faults not known to us when testing the tool. It is important to note, that the *this* object is called $self$, internally, in KeY.

Method Contract	Proof Obligation	Bit-size 3	Bit-size 2
add	Normal Execution	timeout	valid
	Null Reference	valid	valid
	IndexOutOfBounds	valid	valid
	ArrayStoreException	valid	valid
	Pre	valid	valid
	NullReference	valid	valid
	IndexOutOfBounds	valid	valid
get(Normal)	Normal Execution	valid	valid
	NullReference	valid	valid
	IndexOutOfBounds	valid	valid
set(Exceptional)	Normal Execution1	valid	valid
	ClassCastException1	valid	valid
	NullReference1	valid	valid
	IndexOutOfBounds1	valid	valid
	Normal Execution2	valid	valid
	ClassCastException2	valid	valid
	NullReference2	valid	valid
set(Normal)	Normal Execution	valid	valid
	NullReference1	valid	valid
	IndexOutOfBounds1	valid	valid
	ArrayStoreException	valid	valid
	NullReference2	valid	valid
	IndexOutOfBounds2	valid	valid
trimToSize(Normal) 0	Post	timeout	valid
	Pre	valid	valid
	NullReference1	valid	valid
	NullREference2	valid	valid
trimToSize(Normal) 1	NormalExecution	valid	valid
	NullReference	valid	valid

Table 5.1.: Results for closable proof obligations

Method Contract	Bit-size 3	Bit-size 4
Cell::setX	counterexample	counterexample
Saddleback::search	counterexample	counterexample
SimplifiedLL::remove	counterexample	counterexample
ArrayList::indexof	counterexample	timeout
ArrayList::clear	counterexample	timeout
BinarySearch::binarysearch	counterexample	counterexample
Anon::m	counterexample	counterexample
RingBuffer::push	counterexample	counterexample
RingBuffer::pop	counterexample	counterexample

Table 5.2.: Results for not closable proof obligations

5.2.1.1. Method Cell::setX

When running our tool on the open proof obligation of the Cell::setX method described in listing D.1, we obtained the counterexample shown in listing D.2.

The method is a setter for the field x. However, in its contract an assignable clause states, that the method may only modify values from the $footprint$ location set, which is a model field. The $footprint$ of a $Cell$ object contains only the field y and thus a violation of the assignable clause occurs. The counterexample shows the location set $footprint$ for the self object, containing only y.

5.2.1.2. Method Saddleback::search

When running our tool on the open proof obligation of the Saddleback::search method described in listing F.1, we obtained the counterexample shown in listing F.2.

The Saddleback::search method searches for a value inside an two dimensional integer array. The open proof obligation lies on the branch trying to prove the initial validity of the loop invariant. For this reason, when analysing the generated counterexample, we will check for the given array and value if the invariant is true when entering the loop. From the counterexample we can see that the array is $\{\{1, 1, 2\}\}$ and the searched value is 0. Before reaching the invariant we observe that the values for the local variables x and y are 0 and 2 respectively. We notice that the loop invariant contains a decreases statement. When evaluating the term of the decreases statement, we realize that it is equal to -1. For this reason the loop invariant is violated, since decreases terms are required to be larger than or equal to zero at all times.

5.2.1.3. Method SimplifiedLL::remove

When running our tool on the open proof obligation of the SimplifiedLL::remove method described in listing E.1, we obtained the counterexample shown in listing E.2.

The SimplifiedLL::remove method removes the ith Node from a linked list. The SimplifiedLL class has a field $first$, which points to the first node of the list, and a field *size*, giving the number of nodes in the list. Additionally a model field *nodeseq* of the type Sequence has been added in order to reason about the list. The $SimplifiedLL$ class contains an invariant stating, among other things, that all elements of the sequence are of type $Node$. In the method contract of the SimplifiedLL::remove method it is stated that after the call, the *nodeseq* in the new list will be the concatenation of the subsequences from 0 to $i - 1$ and from $i + 1$ to $nodeseq.length - 1$.

Looking at the self object inside the initial heap we see that both the list and the sequence contain the following nodes: $\{\#o4, \#o1, \#o2\}$ and the field *size* has the value of 3. After removing the element with index 2, in the *store_0* heap the list contains as expected the following nodes: $\{\#o4, \#o1\}$, the sequence, however, contains only one node: $\{\#o4\}$. The field *size* has the expected values of 2. Since the class invariants states that all elements of the sequence with indexes from 0 to $size - 1$ are of type $Node$, we can observe that the method violates the class invariant. The reason for the disappearance of the Node $\#o1$ from the sequence lies in the seqSub function definition.

5.2.1.4. Method ArrayList::indexof

When running our tool on the open proof obligation of the ArrayList::indexof method described in listing B.1, we obtained the counterexample shown in listing B.3.

The ArrayList::indexof method returns the first position where the parameter object o can be found, or -1 if the object is not in the list. In the generated counterexample, the self object is the empty Arraylist, and thus the result would always be -1. The reason why this is a counterexample is because it violates the second ensures clause in the method contract. This clause makes the outrageous claim, among other things, that if the object o is not found in the list, then the result must be greater than zero.

5.2.1.5. Method ArrayList::clear

When running our tool on the open proof obligation of the ArrayList::clear method described in listing B.1, we obtained the counterexample shown in listing B.2.

The ArrayList::clear method sets all elements of the *elementData* field of type *Object*[] to *null*, and sets the *size* field to 0. The ArrayList class also contains a model field, *repr*, of type sequence, and the ArrayList::clear method sets this sequence to *seqEmpty*.

When we look at the generated counterexample, we see, that the $self$ object has the empty sequence as the *repr* field, that the *elementData* field points to an array $\{null, null\}$, and that the *size* field has the value 0. It would seem that the ArrayList is already "cleared", yet the specification is violated despite this. Since the *size* attribute is 0, the first while loop has no effect. The loop invariant, however, states that the loop can modify all contents of the *elementData* array. Because the locations which can be modified by the loop are anonymized, we can observe in the *store_0* heap, that this has actually happened, and the contents of the *elementData* array are now $\{\#o2, null\}$. We can now see that the method violates its contract, which claims that all elements of the *elementData* array will be *null* after the method returns.

5.2.2. Specifications with Known Faults

In this section we will present the obtained counterexamples for specifications with faults which we injected.

5.2.2.1. Method BinarySearch::binarysearch

When running our tool on the open proof obligation of the BinarySearch::binarysearch method described in listing A.1, we obtained the counterexample shown in listing A.2.

The code is an iterative implementation of the binary search algorithm. A value v is searched for in a sorted array a. The value is searched in a range starting from index l to r which at the beginning is the entire array. The searched value v is compared with the middle of this range, and, depending on the result of this comparison, either the index of the middle range is returned, or the middle index is set as the left or right margin of the range. This way if the value is not found, the length of the range is halved after each iteration. We changed the way the middle index of the range is calculated in line 21 by dividing through 4 instead of 2. This way the *mid* variable will no longer point to the middle of the range, but right after the first quarter of the range.

The counterexample found by the tool has the input values $a = \{0, 0, 1\}$ and $v = -4$. The values for l and r before entering the loop will be 0 and 2 respectively, and the loop invariant is initially valid. In the first iteration *mid* has the value of 0 and because $a[0]$ is greater than the searched value, *mid* is assigned to r and at the end of the first iteration we will have $l = 0$ and $r = 0$. This violates the loop invariant, which states that $l < r$.

5.2.2.2. Method Anon::m

When running our tool on the open proof obligation of the Anon::m method described in listing C.1, we obtained the counterexample shown in listing C.2.

The *Anon* class has two fields, the *next* field points to another object of the *Anon* type, and the x field of type *int*. The class has an additional method, *modx*, which has no code, but its specification states that it can modify x. The contract which needs to be proven asks that if the value of $this.x = 0$ in the initial state, it will also be 0 after calling the method *modx* on the next object of the next object.

In the counterexample we can observe that in the initial heap, *heap*, the self (KeY name for "this") object is $\#o1$, which points to $\#o4$, which points back to $\#o1$. Thus the method *modx* is actually called on the $self$ object. We can see that in the initial heap the value of x for the self object is 0 and in the heap $heap\_after\_modx$ the value of x for the self object is 2, thus violating the method contract.

5.2.2.3. Method Ringbuffer::push

When running our tool on the open proof obligation of the Ringbuffer::push method described in listing G.1, we obtained the counterexample shown in listing G.2.

The Ringbuffer class is an implementation of a circular list using an array. We have modified the push method by increasing the length of the Ringbuffer by 2 instead of 1 at line 55.

In the counterexample we can see that in the initial heap, *heap*, the value of the *len* field of the $self$ object is 0. In the heap $store\_0$, however, this value changes to 2, as expected. Because in both heaps the length of the *data* object is 1 the class invariant is violated since it requires $0 \leq len \leq data.length$.

5.2.2.4. Method Ringbuffer::pop

When running our tool on the open proof obligation of the Ringbuffer::pop method described in listing G.1, we obtained the counterexample shown in listing G.3.

The Ringbuffer class is an implementation of a circular list using an array. We have modified the pop method by decreasing the length of the Ringbuffer by 2 instead of 1 at line 55.

In the counterexample we can see that in the initial heap, *heap*, the value of the *len* field of the $self$ object is 1. In the heap $store\_1$, however, this value changes to -1, as expected. Thus, the class invariant is violated since it requires $0 \leq len \leq data.length$.

6. Conclusion

6.1. Summary

In this thesis we have designed and implemented a counterexample finder for the KeY verification system. It translates a KeY proof obligations to SMT and hands the resulting SMT specifications to the z3 SMT solver. Only proof obligations written in KeY first order logic(KeYFOL) are supported, meaning that we require proof obligations not to contain modal operators or updates. All KeY types are translated to bounded SMT sorts, thus ensuring decidability.

The top level KeY types *Any*, *Object*, *Heap*, *Field*, *LocSet Sequence*, *Int* and *Bool* are translated to SMT sorts which are aliases of bit-vectors of various sizes. The user can set the bit-sizes of all SMT sorts except *Any*, *Field* and *Heap* which are computed automatically by taking the logarithm of the number of occurrences of the constants of those types. Because the KeY type system is hierarchical, and the SMT one is not, the type hierarchy needs to be encoded. The type *Any* extends its subtypes with additional type bits, and functions for type checking and casting are specified. The type hierarchy of reference types is modelled using the predicates $instance$ and $exactInstance$.

In order to avoid spurious counterexamples, the semantics of KeYFOL interpreted functions and predicates must be preserved. We cannot simply translate the necessary axioms which provide the semantics for these functions, because many axioms imply the existence of certain instances of KeY types, and in order for the SMT specification to remain satisfiable, the bit-sizes of the corresponding SMT sorts would have to be very large and would affect performance. Instead we provide the semantics of the interpreted functions only for the terms on which these functions and predicates are applied. We achieve this using a technique called semantic blasting.

Because we translate KeY integers as bit-vectors the SMT solver may use overflows and generate spurious counterexamples. We provide additional formulae which make sure that a counterexample satisfying the specification will not use overflows.

Because we translate all KeY types to bounded SMT sorts, there are situations in which spurious counterexamples can occur.

We process the counterexample found by the SMT solver and present it in a user friendly way. Currently counterexamples are presented in text form, but, as part of future work, they may be represented in a graphical way.

We have evaluated our tool on several examples in order to see if we get spurious counterexamples when running on valid proof obligations, and if we get counterexamples when running on invalid proof obligations. We have also shown how the found counterexamples can help the user identify the fault.

6.2. Related Work

6.2.1. The Previous Translation to SMT

KeY already provides a translation to the SMT-LIB format. This old translation, however, serves the purpose of proving proof obligations. For this reason the used sorts are unbounded. The type system is modelled using a single SMT-Sort u with $typeOf$ and $exatInstanceOf$ predicates for each KeY sort. Compared to our translation, the type hierarchy is underspecified, it is only stated that an object of a type T is also type of the parents of T. The underspecification renders this translation of little use when searching for counterexamples, because it does not assert what types an object cannot be. Thus we can get counterexamples with each objects being of all types.

The previous translation does not provide a semantic blasting mechanism, the only way to preserve the semantics of KeY functions and predicates is to translate the taclets which specify their semantics. The user can choose which taclets to translate, and he must know which taclets are actually needed, otherwise unneeded formulae will be added to the specification. On the other hand, if the user does not choose the necessary taclets, the functions and predicates are left uninterpreted, which can cause spurious counterexamples. Semantic blasting automatically specifies the semantics only for the functions and predicates needed for the arguments occurring in the proof obligation, thus simplifying the complexity of the specification.

Although our translation is currently better suited for counterexample finding, we could adapt it to fulfill the goal of proving proof obligations as well. Having a more exact specification, we would be able to prove more proof obligations than the old translation, because we restrict the space in which counterexamples may be found.

6.2.2. Nitpick

Nitpick [BN10] is a counterexample generator for the Isabelle [NPW02] proof assistant, serving a similar purpose as our tool. It translates an Isabelle conjecture from higher order logic (HOL) to relational first order logic (RFOL), which is then checked with Alloy's [Jac02] backend, Kodkod [TJ07]. Kodkod translates the problem to SAT using sophisticated simplification techniques like symmetry breaking.

Nitpick translates Isabelle's functions to the corresponding built in Kodkod functions when possible, avoiding the translation of the semantics of the HOL functions. This is similar to how our tool uses the built in SMT functions for the boolean and bitvector types. Since Kodkod uses SAT and our tool uses an SMT solver, we have real decision procedures for some of the SMT built-in functions, which can improve performance.

Another similarity to our tool is that all types are given bounds. For infinite types, like integers, only a finite subdomain is considered.

A difference, however, results from the purposes of the KeY and Isabelle tools. While KeY specializes in proving properties of Java programs, Isabelle is a more general prover. As such, the counterexamples found by our tool present the state before and after executing a Java program in a user friendly way. We treat constant functions differently than the select functions when presenting the counterexample to the user. Because of the more general purpose of Isabelle, Nitpick treats all functions equally, making counterexamples for proof obligations similar to ours more difficult for the user to read.

6.2.3. Dynamite

Dynamite [FPM07, MLPF10] is a tool for proving Alloy [Jac02] specifications using the PVS [ORS92] theorem prover. PVS uses a sequent calculus for a higher order logic. In order to support Alloy specifications, PVS was extended with a complete calculus for Alloy. Similarly to our tool, Dynamite uses a bounded verification tool,the Alloy Analizer, in order to check hypotheses and lemmas introduced by the user, thus lowering the chances of introducing an invalid formula. A difference between our tool and Dynamite lies in the bounded verification technique used. While we use an SMT solver, which provides decision procedures for some built in functions, Dynamite uses the Alloy Analizer which is based on a SAT solver. A further difference lies in the logics used by the two approaches. Dynamite uses a higher order logic, while our tool, in combination with KeY, uses JavaDL. Thus, as in the case of Nitpick, our tool is adapted for the context of verifying specified Java programs, while Dynamite has a more general purpose.

6.2.4. Lightweight Verification Tools for Java

Other verification tools which employ SMT solvers for specified Java programs have been developed. Whereas our tool serves as an assistant to a larger verification system, KeY, these tools run independently. One such tool is Esc/Java 2 [CK05], which can generates an SMT specification from a Java program specified with JML. It uses unbounded sorts, thus being undecidable. Additionally, because the purpose of Esc/Java 2 is to be a lightweight verification tool, soundness is sometimes sacrificed for comfort. For example, loops are not specified using loop invariants, but they are unwinded only once.

Another static verification tool for specified Java programs is InspectJ [LNT12], which also generates an SMT specification. The specification uses only bounded sorts, which makes it decidable. InspectJ offers a more limited support of Java language constructs, and JML statements. For example, it does not support interfaces and abstract classes, loop invariants or arithmetic overflow checking.

6.3. Future Work

There are several ways in which this project could be improved. We can implement better ways for showing the counterexample to the user. A possible improvement of the current way of presenting counterexample would be to find out which objects, location sets, and sequences are actually needed for the counterexample, and display only those.

An additional way to present counterexamples would be to generate a UML object diagram based on the model. This diagram would have to differ from a standard UML object diagram, because it would need to display the contents of heaps, location sets and sequences, which are not supported by standard UML. A possible representation of a mock counterexample as a UML object diagram is shown in figure 6.1.

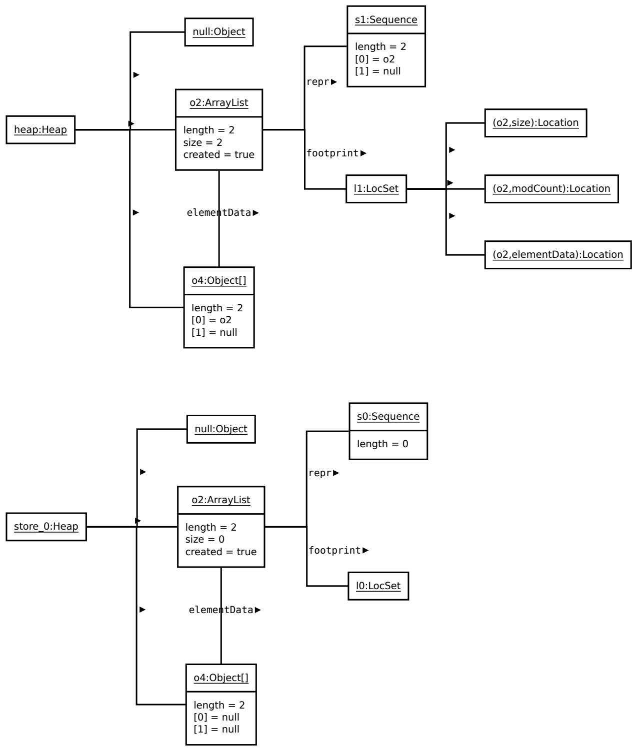

Figure 6.1.: Representation of a mock counterexample as a UML object diagram

A third way to present the counterexample is through a tree representation. All constants, heaps, objects, locations sets and sequences would be nodes in the tree. When the user clicks on a node, the attributes of that node will be shown to the user. Clicking on an attribute node would show the attributes of the clicked node. A possible representation of a counterexample as a tree is shown in figure 6.2.

A further way to improve the translation to SMT would be to provide support for unbounded sorts. The only thing preventing an unbounded translation is the SMT type system, which uses type bits and bit-vector extraction and concatenation operations to cast between the sorts *Any* and its subsorts. In order to support unbounded sorts, we need to provide a special specification in the unbounded case for the functions $Any2S$, $S2Any$ and isS for each SMT sort S, subtype of Any. The advantage towards the previous translation to SMT would be the fact that by using semantic blasting we get rid of a large number of unnecessary quantifiers. Furthermore because our type hierarchy is more precisely specified we further restrict the search space of the SMT solver.

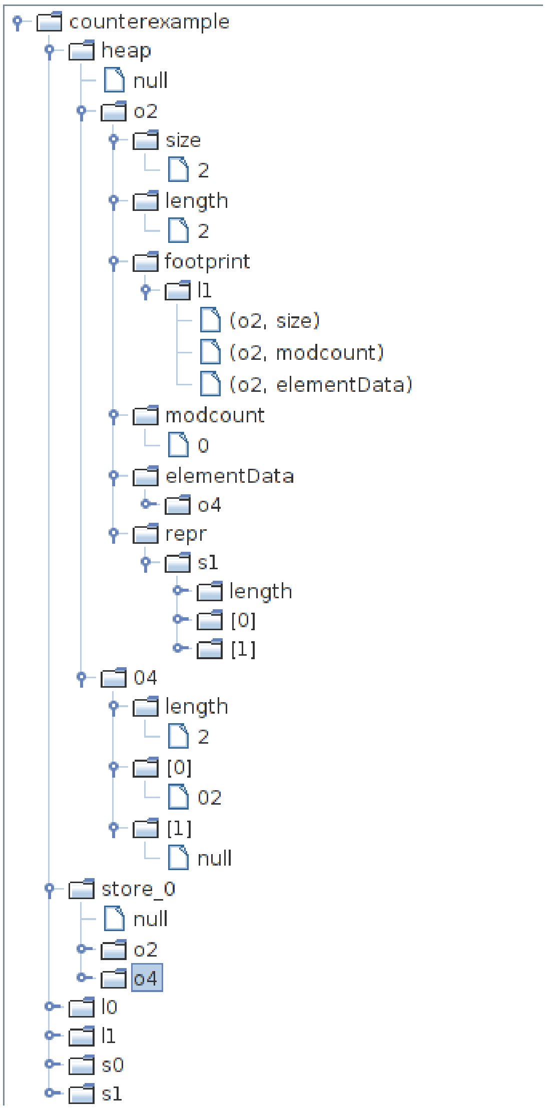

Figure 6.2.: Representation of a counterexample as a tree

An additional improvement would be the possibility of combining semantic blasting with the translation of rules. The user could then chose which rules to translate and for which to use semantic blasting. This way we could support recursive functions and avoid spurious counterexamples caused by the lack of certain rules as described in Section 3.7. However, translating rules may affect performance as described in Section 3.3.

Last but not least we could use the tool to generate test cases from our counterexamples. It is fairly easy to determine the input of the program from the counterexample, and we can use that input to generate a test case which will fail if the counterexample is correct.

Bibliography

[Bec01] Bernhard Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, LNCS 2041, pages 6–24. Springer, 2001.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-Oriented Software - The KeY Approach.* Springer-Verlag, 2007.

[BN10] Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In *Proceedings of the First International Conference on Interactive Theorem Proving*, ITP'10, pages 131–146, Berlin, Heidelberg, 2010. Springer-Verlag.

[BST12] Clark Barrett, Aaron Stump, and Cesare Tinelli. The smt-lib standard version 2.0. `http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.0-r12.09.09.pdf`, 2012. Accessed: 2013-12-16.

[CK05] DavidR. Cok and JosephR. Kiniry. Esc/java2: Uniting esc/java and jml. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer Berlin Heidelberg, 2005.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[FPM07] Marcelo F. Frias, Carlos G. Lopez Pombo, and Mariano M. Moscato. Alloy analyzer+pvs in the analysis and verification of alloy specifications. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'07, pages 587–601, Berlin, Heidelberg, 2007. Springer-Verlag.

[HKT84] David Harel, Dexter Kozen, and Jerzy Tiuryn. Dynamic logic. In *Handbook of Philosophical Logic*, pages 497–604. MIT Press, 1984.

[Jac02] Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, April 2002.

[LNT12] Tianhai Liu, Michael Nagel, and Mana Taghdiri. Bounded program verification using an smt solver: A case study. In *5th International Conference on Software Testing, Verification and Validation (ICST)*, pages 101–110, April 2012.

[MLPF10] Mariano Moscato, Carlos Lopez Pombo, and Marcelo Frias. Dynamite 2.0: New features based on unsat-core extraction to improve verification of software requirements, 2010. 7th International Colloquium, Natal, Rio Grande do Norte, Brazil, September 1-3, 2010.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[ORS92] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[TJ07] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *In Tools and Algorithms for Construction and Analysis of Systems (TACAS*, pages 632–647. Wiley, 2007.

[Wei11] Benjamin Weiß. *Deductive verification of object-oriented software : dynamic frames, dynamic Logic and predicate abstraction.* PhD thesis, Karlsruhe, 2011.

7. Appendix

A. Binary Search

A.1. Specified Java Code

```
class BinarySearch {

    /*@ public normal_behaviour
      @  requires (\forall int x; (\forall int y; 0 <= x && x < y && y <
         a.length; a[x] <= a[y]));
      @  ensures ((\exists int x; 0 <= x && x < a.length; a[x] == v) ?
         a[\result] == v : \result == -1);
      @*/
    static /*@pure@*/ int search(int[] a, int v) {
        int l = 0;
        int r = a.length - 1;

        if(a.length == 0) return -1;
        if(a.length == 1) return a[0] == v ? 0 : -1;

        /*@ loop_invariant 0 <= l && l < r && r < a.length
          @               && (\forall int x; 0 <= x && x < l; a[x] < v)
          @               && (\forall int x; r < x && x < a.length; v <
             a[x]);
          @ assignable \nothing;
          @ decreases r - l;
          @*/
        while(r > l + 1) {
            int mid = l + (r - l) / 4;
            if(a[mid] == v) {
               return mid;
            } else if(a[mid] > v) {
               r = mid;
            } else {
               l = mid;
```

```
            }
        }
        if(a[l] == v) return l;
        if(a[r] == v) return r;
        return -1;
    }
}
```

A.2. Counterexample for BinarySearch::binarySearch

```
Constants
-----------
heap = #h0
v = -4
seqEmpty = #s0
l_0 = 0
|java.lang.Object::<created>| = #f8
a = #o2
anon_heap_loop = #h0
empty = #l6
seqGetOutside = #a40
r_0 = 2
null = #o0


Heaps
-----------
Heap heap
   Object #o0/null

   Object #o1
      length = 0
      type =java.util.ListIterator
      exactInstance =false
      |java.lang.Object::<created>| = true
      classInvariant = false

   Object #o2/a
      length = 3
      type =int[]
      exactInstance =true
      |java.lang.Object::<created>| = true
      classInvariant = false
      [0] = 0
      [1] = 0
      [2] = 1

   Object #o3
      length = 0
      type =int[]
      exactInstance =true
```

```
      |java.lang.Object::<created>| = true
      classInvariant = false

   Object #o4
      length = 0
      type =java.util.ListIterator
      exactInstance =false
      |java.lang.Object::<created>| = false
      classInvariant = false

   Object #o5
      length = 0
      type =java.util.ListIterator
      exactInstance =false
      |java.lang.Object::<created>| = false
      classInvariant = false

   Object #o6
      length = 0
      type =int[]
      exactInstance =true
      |java.lang.Object::<created>| = true
      classInvariant = false

   Object #o7
      length = 0
      type =int[]
      exactInstance =true
      |java.lang.Object::<created>| = true
      classInvariant = false


Heap anon_heap_loop
   Object #o0/null

   Object #o1
      length = 0
      type =java.util.ListIterator
      exactInstance =false
      |java.lang.Object::<created>| = true
      classInvariant = false

   Object #o2/a
      length = 3
      type =int[]
      exactInstance =true
      |java.lang.Object::<created>| = true
      classInvariant = false
      [0] = 0
      [1] = 0
      [2] = 1
```

```
   Object #o3
      length = 0
      type =int[]
      exactInstance =true
      |java.lang.Object::<created>| = true
      classInvariant = false

   Object #o4
      length = 0
      type =java.util.ListIterator
      exactInstance =false
      |java.lang.Object::<created>| = false
      classInvariant = false

   Object #o5
      length = 0
      type =java.util.ListIterator
      exactInstance =false
      |java.lang.Object::<created>| = false
      classInvariant = false

   Object #o6
      length = 0
      type =int[]
      exactInstance =true
      |java.lang.Object::<created>| = true
      classInvariant = false

   Object #o7
      length = 0
      type =int[]
      exactInstance =true
      |java.lang.Object::<created>| = true
      classInvariant = false



Location Sets
-----------
#l0 = {}
#l1 = {}
#l2 = {}
#l3 = {}
#l4 = {}
#l5 = {}
#l6 = {}
#l7 = {}

Sequences
-----------
Seq: #s0/seqEmpty
Length: 0
```

```

Seq: #s1
Length: 0

Seq: #s2
Length: 0

Seq: #s3
Length: 0

Seq: #s4
Length: 0

Seq: #s5
Length: 0

Seq: #s6
Length: 0

Seq: #s7
Length: 0
```

B. ArrayList

B.1. Specified Java Code

```java

class SelfArrays {

  /*@ public normal_behavior
    @   requires original != null;
    @   requires newLength >= 0;
    @   requires \typeof(original) == \type(java.lang.Object[]);
    @   ensures \typeof(original) == \typeof(\result);
    @   ensures \typeof(\result) == \type(java.lang.Object[]);
    @   ensures newLength < original.length ==>
          (\forall int i; 0 <= i && i < newLength; \result[i] ==
             original[i]);
    @   ensures newLength >= original.length ==>
          (\forall int i; 0 <= i && i < original.length; \result[i] ==
             original[i]);
    @   ensures newLength > original.length ==>
          (\forall int i; original.length <= i && i < newLength;
             \result[i] == null);
    @   ensures \result.length == newLength;
    @   ensures \fresh(\result);
    @   ensures \result != null;
    @   assignable \nothing;
    @ also
    @ public exceptional_behavior
    @   requires (newLength < 0) || (original == null);
```

```
    @   signals (NegativeArraySizeException) newLength < 0;
    @   signals (NullPointerException) original == null;
    @   signals_only NegativeArraySizeException, NullPointerException;
    @   assignable \nothing;
    @*/
  native public /*@ helper nullable @*/ Object[] copyOf(/*@ nullable @*/ Object[] original, int newLength);
}

public class ArrayList {

  /*@ public model instance \locset footprint;
    @ public accessible \inv: footprint;
    @ public accessible footprint: footprint;
    @
    @ public nullable ghost instance \seq repr;
    @ public model instance int seqLength;
    @ public accessible seqLength: footprint;
    @
    @ represents footprint = elementData, elementData[*], size, modCount, repr;
    @
    @ public instance invariant (\forall int i; 0 <= i && i < repr.length;
          repr[i] == elementData[i]);
    @ public represents seqLength = size;
    @
    @ public instance invariant size == repr.length;
    @ public instance invariant seqLength <= elementData.length;
    @ public instance invariant \typeof(elementData) == \type(java.lang.Object[]);
    @ public instance invariant modCount >= 0;
    @ public instance invariant size >= 0;
    @*/

  private Object[] /*@ spec_public nullable @*/ elementData;
  //@ public instance invariant elementData != null;

  private SelfArrays selfArrays;

  protected int /*@ spec_public @*/ modCount = 0;
  /*@ spec_public @*/ protected int size;

  //////////////
  /* Method 1 */
  //////////////

  /*@ public normal_behavior
    @   requires initialCapacity >= 0;
    @   ensures elementData.length == initialCapacity;
    @   ensures (\forall int i; 0 <= i && i < seqLength; elementData[i] == null);
    @   ensures repr == \seq_empty;
```

```
    @   ensures \fresh(footprint);
    @   assignable footprint;
    @*/
  public ArrayList(int initialCapacity) {
    if (initialCapacity < 0)
      throw new IllegalArgumentException();

    this.elementData = new Object[initialCapacity];
    //@ set repr = \seq_empty;
    {}
  }

  //////////////
  /* Method 2 */
  //////////////

  /*@ public normal_behavior
    @   requires index >= 0 && index < seqLength;
    @   ensures \result == repr[index];
    @   assignable \strictly_nothing;
    @ also
    @ public exceptional_behavior
    @   requires index < 0 || index >= seqLength;
    @   signals (IndexOutOfBoundsException) true;
    @   assignable \nothing;
    @*/
  /*@ nullable @*/ public Object get(int index) {
    if (index >= size || index < 0)
      throw new IndexOutOfBoundsException();

    return elementData[index];
  }

  //////////////
  /* Method 3 */
  //////////////

  /*@ public normal_behavior
    @   requires size < elementData.length;
    @   ensures modCount == \old(modCount) + 1;
    @   ensures elementData.length == size;
    @   ensures repr == \old(repr);
    @   assignable elementData, modCount;
    @ also
    @ public normal_behavior
    @   requires size >= elementData.length;
    @   ensures modCount == \old(modCount) + 1;
    @   ensures repr == \old(repr);
    @   assignable modCount;
    @*/
  public void trimToSize() {
    modCount++;
```

```
    int oldCapacity = elementData.length;
    if (size < oldCapacity) {
      elementData = selfArrays.copyOf(elementData, size);
    }
  }

  //////////////
  /* Method 4 */
  //////////////

  /*@ public normal_behavior
    @   ensures modCount == \old(modCount) + 1;
    @   ensures size == 0;
    @   ensures repr == \seq_empty;
    @   ensures (\forall int i; 0 <= i && i < elementData.length;
        elementData[i] == null);
    @   assignable elementData[*], repr, size, modCount;
    @*/
  public void clear() {
    modCount++;

    int i = 0;
    /*@ loop_invariant 0 <= i && i <= size;
      @ loop_invariant (\forall int j; 0 < j && j < i; elementData[j] ==
          null);
      @ assignable elementData[*];
      @ decreasing size - i;
      @*/
    while(i < size) {
      elementData[i] = null;
      i++;
    }

    //@ set repr = \seq_empty;
    size = 0;
  }

  //////////////
  /* Method 5 */
  //////////////

  /*@ public normal_behavior
    @   requires \typeof(element) == \type(Object);
    @   requires index >= 0 && index < seqLength;
    @   ensures repr[index] == element;
    @   ensures \result == \old(repr[index]);
    @   ensures \new_elems_fresh(footprint);
    @   assignable footprint;
    @ also
    @ public exceptional_behavior
    @   requires index < 0 || index >= seqLength;
```

```
    @   signals (IndexOutOfBoundsException) index < 0 || index >=
        seqLength;
    @   signals_only IndexOutOfBoundsException;
    @   assignable \nothing;
    @*/
  public /*@ nullable @*/ Object set(int index, /*@ nullable @*/ Object
      element) {
    if (index >= size || index < 0)
      throw new IndexOutOfBoundsException();

    Object oldValue = elementData[index];
    elementData[index] = element;

    /*@ set repr = \seq_concat(
          \seq_concat(\seq_sub(repr, 0, index), \seq_singleton(element)),
          \seq_sub(repr, index + 1, seqLength)
        );
      @*/

    return oldValue;
  }

  //////////////
  /* Method 6 */
  //////////////

  /*@ public normal_behavior
    @   requires \typeof(e) == \type(java.lang.Object) && e.\inv;
    @   ensures seqLength == \old(seqLength) + 1;
    @   ensures (\forall int i; 0 <= i && i < seqLength-1;
          repr[i] == \old(repr[i]));
    @   ensures repr[seqLength-1] == e;
    @   ensures \result;
    @   assignable footprint;
    @*/
  boolean add( /*@ nullable @*/ Object e) {
    elementData = selfArrays.copyOf(elementData, size + 1);
    elementData[size++] = e;
    //@ set repr = \seq_concat(repr, \seq_singleton(e));
    {}
    return true;
  }

  //////////////
  /* Method 7 */
  //////////////

  /*@ public normal_behavior
    @   requires \typeof(o) == \type(Object) && o.\inv;
    @   ensures (\forall int k; 0 <= k && k <= seqLength-1; repr[k] != o)
          ==> \result == -1;
    @   ensures !(\exists int k; 0 <= k && k <= seqLength-1; repr[k] == o)
```

```
        ==> (\result >= 0 && \result < seqLength && repr[\result] == o);
    @   ensures !(\exists int k; 0 <= k && k < \result; repr[k] == o);
    @   assignable \nothing;
    @*/
  public int indexOf( /*@ nullable @*/ Object o) {

    if (o == null) {

      /*@ loop_invariant 0 <= i && i <= size;
        @ loop_invariant (\forall int j; 0 <= j && j < i; repr[j] != o);
        @ assignable \strictly_nothing;
        @ decreasing size - i;
        @*/
      for(int i = 0; i < size; i++)
        if(elementData[i] == null)
          return i;
    } else {
      /*@ loop_invariant 0 <= i && i <= size;
        @ loop_invariant (\forall int j; 0 <= j && j < i; repr[j] != o);
        @ assignable \nothing;
        @ decreasing size - i;
        @*/
      for (int i = 0; i < size; i++)
        if(o == elementData[i])
         return i;
    }
    return -1;
  }

}
```

B.2. Counterexample for ArrayList::clear

```
Constants
-----------
|arrlist.ArrayList::size| = #f10
heap = #h2
store_0 = #h0
seqEmpty = #s0
|arrlist.ArrayList::modCount| = #f11
|arrlist.ArrayList::elementData| = #f13
i_0 = 0
|arrlist.ArrayList::repr| = #f8
|arrlist.ArrayList::selfArrays| = #f9
|java.lang.Object::<created>| = #f12
anon_heap_loop = #h0
empty = #l0
seqGetOutside = #a0
self = #o2
null = #o0
allFields_0 = #l2
```

```


Heaps
-----------
Heap heap
  Object #o0/null

  Object #o1
    length = 2
    type =arrlist.SelfArrays
    exactInstance =false
    |java.lang.Object::<created>| = false
    classInvariant = true
    |arrlist.ArrayList::$footprint| = #l0
    |arrlist.ArrayList::$seqLength| = 0

  Object #o2/self
    length = 2
    type =arrlist.ArrayList
    exactInstance =true
    |arrlist.ArrayList::elementData| = #o4
    |arrlist.ArrayList::modCount| = 0
    |arrlist.ArrayList::repr| = #s0
    |arrlist.ArrayList::selfArrays| = #o5
    |arrlist.ArrayList::size| = 0
    |java.lang.Object::<created>| = true
    classInvariant = true
    |arrlist.ArrayList::$footprint| = #l1
    |arrlist.ArrayList::$seqLength| = 0

  Object #o3
    length = 2
    type =java.lang.Object
    exactInstance =false
    |java.lang.Object::<created>| = false
    classInvariant = true
    |arrlist.ArrayList::$footprint| = #l0
    |arrlist.ArrayList::$seqLength| = 0

  Object #o4
    length = 2
    type =java.lang.Object[]
    exactInstance =true
    |java.lang.Object::<created>| = true
    classInvariant = true
    |arrlist.ArrayList::$footprint| = #l0
    |arrlist.ArrayList::$seqLength| = 0
    [0] = #o0/null
    [1] = #o0/null

  Object #o5
    length = 2
```

```
      type =arrlist.SelfArrays
      exactInstance =false
      |java.lang.Object::<created>| = true
      classInvariant = true
      |arrlist.ArrayList::$footprint| = #l0
      |arrlist.ArrayList::$seqLength| = 0

   Object #o6
      length = 2
      type =java.util.Set
      exactInstance =false
      |java.lang.Object::<created>| = false
      classInvariant = true
      |arrlist.ArrayList::$footprint| = #l0
      |arrlist.ArrayList::$seqLength| = 0

   Object #o7
      length = 2
      type =java.lang.Object
      exactInstance =false
      |java.lang.Object::<created>| = false
      classInvariant = true
      |arrlist.ArrayList::$footprint| = #l0
      |arrlist.ArrayList::$seqLength| = 0


Heap anon_heap_loop
   Object #o0/null

   Object #o1
      length = 2
      type =arrlist.SelfArrays
      exactInstance =false
      |java.lang.Object::<created>| = true
      classInvariant = true
      |arrlist.ArrayList::$footprint| = #l0
      |arrlist.ArrayList::$seqLength| = 0

   Object #o2/self
      length = 2
      type =arrlist.ArrayList
      exactInstance =true
      |arrlist.ArrayList::elementData| = #o4
      |arrlist.ArrayList::modCount| = 1
      |arrlist.ArrayList::repr| = #s0
      |arrlist.ArrayList::selfArrays| = #o5
      |arrlist.ArrayList::size| = 0
      |java.lang.Object::<created>| = true
      classInvariant = true
      |arrlist.ArrayList::$footprint| = #l4
      |arrlist.ArrayList::$seqLength| = 0
```

```
  Object #o3
    length = 2
    type =java.lang.Object
    exactInstance =false
    |java.lang.Object::<created>| = false
    classInvariant = true
    |arrlist.ArrayList::$footprint| = #l0
    |arrlist.ArrayList::$seqLength| = 0

  Object #o4
    length = 2
    type =java.lang.Object[]
    exactInstance =true
    |java.lang.Object::<created>| = true
    classInvariant = true
    |arrlist.ArrayList::$footprint| = #l0
    |arrlist.ArrayList::$seqLength| = 0
    [0] = #o4
    [1] = #o4

  Object #o5
    length = 2
    type =arrlist.SelfArrays
    exactInstance =false
    |java.lang.Object::<created>| = true
    classInvariant = true
    |arrlist.ArrayList::$footprint| = #l0
    |arrlist.ArrayList::$seqLength| = 0

  Object #o6
    length = 2
    type =java.util.Set
    exactInstance =false
    |java.lang.Object::<created>| = false
    classInvariant = true
    |arrlist.ArrayList::$footprint| = #l0
    |arrlist.ArrayList::$seqLength| = 0

  Object #o7
    length = 2
    type =java.lang.Object
    exactInstance =false
    |java.lang.Object::<created>| = false
    classInvariant = true
    |arrlist.ArrayList::$footprint| = #l0
    |arrlist.ArrayList::$seqLength| = 0


Heap store_0
  Object #o0/null

  Object #o1
```

```
    length = 2
    type =arrlist.SelfArrays
    exactInstance =false
    |java.lang.Object::<created>| = true
    classInvariant = true
    |arrlist.ArrayList::$footprint| = #l0
    |arrlist.ArrayList::$seqLength| = 0

  Object #o2/self
    length = 2
    type =arrlist.ArrayList
    exactInstance =true
    |arrlist.ArrayList::elementData| = #o4
    |arrlist.ArrayList::modCount| = 1
    |arrlist.ArrayList::repr| = #s0
    |arrlist.ArrayList::selfArrays| = #o5
    |arrlist.ArrayList::size| = 0
    |java.lang.Object::<created>| = true
    classInvariant = true
    |arrlist.ArrayList::$footprint| = #l4
    |arrlist.ArrayList::$seqLength| = 0

  Object #o3
    length = 2
    type =java.lang.Object
    exactInstance =false
    |java.lang.Object::<created>| = false
    classInvariant = true
    |arrlist.ArrayList::$footprint| = #l0
    |arrlist.ArrayList::$seqLength| = 0

  Object #o4
    length = 2
    type =java.lang.Object[]
    exactInstance =true
    |java.lang.Object::<created>| = true
    classInvariant = true
    |arrlist.ArrayList::$footprint| = #l0
    |arrlist.ArrayList::$seqLength| = 0
    [0] = #o4
    [1] = #o4

  Object #o5
    length = 2
    type =arrlist.SelfArrays
    exactInstance =false
    |java.lang.Object::<created>| = true
    classInvariant = true
    |arrlist.ArrayList::$footprint| = #l0
    |arrlist.ArrayList::$seqLength| = 0

  Object #o6
```

```
      length = 2
      type =java.util.Set
      exactInstance =false
      |java.lang.Object::<created>| = false
      classInvariant = true
      |arrlist.ArrayList::$footprint| = #l0
      |arrlist.ArrayList::$seqLength| = 0

   Object #o7
      length = 2
      type =java.lang.Object
      exactInstance =false
      |java.lang.Object::<created>| = false
      classInvariant = true
      |arrlist.ArrayList::$footprint| = #l0
      |arrlist.ArrayList::$seqLength| = 0



Location Sets
-----------
#l0 = {}
#l1 = {(#o2/self, |arrlist.ArrayList::repr|), (#o2/self,
    |arrlist.ArrayList::size|), (#o2/self, |arrlist.ArrayList::modCount|),
    (#o2/self, |arrlist.ArrayList::elementData|), (#o4, [0]), (#o4, [1]),
    (#o4, [2]), (#o4, [3]), (#o4, |arrlist.ArrayList::repr|), (#o4,
    |arrlist.ArrayList::selfArrays|), (#o4, |arrlist.ArrayList::size|),
    (#o4, |arrlist.ArrayList::modCount|), (#o4,
    |java.lang.Object::<created>|), (#o4,
    |arrlist.ArrayList::elementData|)}
#l2 = {(#o4, [0]), (#o4, [1]), (#o4, [2]), (#o4, [3]), (#o4,
    |arrlist.ArrayList::repr|), (#o4, |arrlist.ArrayList::selfArrays|),
    (#o4, |arrlist.ArrayList::size|), (#o4,
    |arrlist.ArrayList::modCount|), (#o4, |java.lang.Object::<created>|),
    (#o4, |arrlist.ArrayList::elementData|)}
#l3 = {(#o0/null, [0]), (#o0/null, [1]), (#o0/null, [2]), (#o0/null,
    [3]), (#o0/null, |arrlist.ArrayList::repr|), (#o0/null,
    |arrlist.ArrayList::selfArrays|), (#o0/null,
    |arrlist.ArrayList::size|), (#o0/null, |arrlist.ArrayList::modCount|),
    (#o0/null, |java.lang.Object::<created>|), (#o0/null,
    |arrlist.ArrayList::elementData|)}
#l4 = {(#o2/self, |arrlist.ArrayList::repr|), (#o2/self,
    |arrlist.ArrayList::size|), (#o2/self, |arrlist.ArrayList::modCount|),
    (#o2/self, |arrlist.ArrayList::elementData|), (#o4, [0]), (#o4, [1]),
    (#o4, [2]), (#o4, [3]), (#o4, |arrlist.ArrayList::repr|), (#o4,
    |arrlist.ArrayList::selfArrays|), (#o4, |arrlist.ArrayList::size|),
    (#o4, |arrlist.ArrayList::modCount|), (#o4,
    |java.lang.Object::<created>|), (#o4,
    |arrlist.ArrayList::elementData|)}
#l5 = {}
#l6 = {(#o0/null, [0]), (#o0/null, [1]), (#o0/null, [2]), (#o0/null,
    [3]), (#o0/null, |arrlist.ArrayList::repr|), (#o0/null,
```

```
    |arrlist.ArrayList::selfArrays|), (#o0/null,
    |arrlist.ArrayList::size|), (#o0/null, |arrlist.ArrayList::modCount|),
    (#o0/null, |java.lang.Object::<created>|), (#o0/null,
    |arrlist.ArrayList::elementData|)}
#l7 = {}

Sequences
-----------
Seq: #s0/seqEmpty
Length: 0

Seq: #s1
Length: 0

Seq: #s2
Length: 0

Seq: #s3
Length: 0

Seq: #s4
Length: 0

Seq: #s5
Length: 0

Seq: #s6
Length: 0

Seq: #s7
Length: 0
```

B.3. Counterexample for ArrayList::indexOf

```
Constants
-----------
seqEmpty = #s0
|java.lang.Object::<created>| = #f12
anon_1 = #h0
anon_2 = #h0
o = #o2
anon_0 = #h0
empty_0 = #l0
empty_1 = #l0
heap = #h0
|arrlist.ArrayList::size| = #f10
self_0 = #o1
|arrlist.ArrayList::modCount| = #f11
|arrlist.ArrayList::elementData| = #f13
i_0 = 0
|arrlist.ArrayList::repr| = #f8
```

```
|arrlist.ArrayList::selfArrays| = #f9
empty_2 = #l0
anon_heap_loop = #h0
empty = #l6
anon_3 = #h0
seqGetOutside = #a40
null = #o0


Heaps
-----------
Heap heap
   Object #o0/null

   Object #o1/self_0
      length = 0
      type =arrlist.ArrayList
      exactInstance =true
      |arrlist.ArrayList::elementData| = #o4
      |arrlist.ArrayList::modCount| = 1
      |arrlist.ArrayList::repr| = #s0
      |arrlist.ArrayList::selfArrays| = #o6
      |arrlist.ArrayList::size| = 0
      |java.lang.Object::<created>| = true
      classInvariant = true
      |arrlist.ArrayList::$footprint| = #l2
      |arrlist.ArrayList::$seqLength| = 0

   Object #o2/o
      length = 0
      type =java.lang.Object
      exactInstance =false
      |java.lang.Object::<created>| = true
      classInvariant = true
      |arrlist.ArrayList::$footprint| = #l2
      |arrlist.ArrayList::$seqLength| = 0

   Object #o3
      length = 0
      type =java.lang.Object
      exactInstance =false
      |java.lang.Object::<created>| = true
      classInvariant = true
      |arrlist.ArrayList::$footprint| = #l2
      |arrlist.ArrayList::$seqLength| = 0

   Object #o4
      length = 0
      type =java.lang.Object[]
      exactInstance =true
      |java.lang.Object::<created>| = true
      classInvariant = true
```

```
        |arrlist.ArrayList::$footprint| = #l2
        |arrlist.ArrayList::$seqLength| = 0

    Object #o5
        length = 0
        type =java.lang.Object
        exactInstance =false
        |java.lang.Object::<created>| = true
        classInvariant = true
        |arrlist.ArrayList::$footprint| = #l2
        |arrlist.ArrayList::$seqLength| = 0

    Object #o6
        length = 0
        type =arrlist.SelfArrays
        exactInstance =false
        |java.lang.Object::<created>| = true
        classInvariant = true
        |arrlist.ArrayList::$footprint| = #l2
        |arrlist.ArrayList::$seqLength| = 0

    Object #o7
        length = 0
        type =java.lang.Object
        exactInstance =false
        |java.lang.Object::<created>| = true
        classInvariant = true
        |arrlist.ArrayList::$footprint| = #l2
        |arrlist.ArrayList::$seqLength| = 0


Heap anon_heap_loop
    Object #o0/null

    Object #o1/self_0
        length = 0
        type =arrlist.ArrayList
        exactInstance =true
        |arrlist.ArrayList::elementData| = #o4
        |arrlist.ArrayList::modCount| = 1
        |arrlist.ArrayList::repr| = #s0
        |arrlist.ArrayList::selfArrays| = #o6
        |arrlist.ArrayList::size| = 0
        |java.lang.Object::<created>| = true
        classInvariant = true
        |arrlist.ArrayList::$footprint| = #l2
        |arrlist.ArrayList::$seqLength| = 0

    Object #o2/o
        length = 0
        type =java.lang.Object
        exactInstance =false
```

```
        |java.lang.Object::<created>| = true
        classInvariant = true
        |arrlist.ArrayList::$footprint| = #l2
        |arrlist.ArrayList::$seqLength| = 0

    Object #o3
        length = 0
        type =java.lang.Object
        exactInstance =false
        |java.lang.Object::<created>| = true
        classInvariant = true
        |arrlist.ArrayList::$footprint| = #l2
        |arrlist.ArrayList::$seqLength| = 0

    Object #o4
        length = 0
        type =java.lang.Object[]
        exactInstance =true
        |java.lang.Object::<created>| = true
        classInvariant = true
        |arrlist.ArrayList::$footprint| = #l2
        |arrlist.ArrayList::$seqLength| = 0

    Object #o5
        length = 0
        type =java.lang.Object
        exactInstance =false
        |java.lang.Object::<created>| = true
        classInvariant = true
        |arrlist.ArrayList::$footprint| = #l2
        |arrlist.ArrayList::$seqLength| = 0

    Object #o6
        length = 0
        type =arrlist.SelfArrays
        exactInstance =false
        |java.lang.Object::<created>| = true
        classInvariant = true
        |arrlist.ArrayList::$footprint| = #l2
        |arrlist.ArrayList::$seqLength| = 0

    Object #o7
        length = 0
        type =java.lang.Object
        exactInstance =false
        |java.lang.Object::<created>| = true
        classInvariant = true
        |arrlist.ArrayList::$footprint| = #l2
        |arrlist.ArrayList::$seqLength| = 0


Heap anon_3
```

```
Object #o0/null

Object #o1/self_0
   length = 0
   type =arrlist.ArrayList
   exactInstance =true
   |arrlist.ArrayList::elementData| = #o4
   |arrlist.ArrayList::modCount| = 1
   |arrlist.ArrayList::repr| = #s0
   |arrlist.ArrayList::selfArrays| = #o6
   |arrlist.ArrayList::size| = 0
   |java.lang.Object::<created>| = true
   classInvariant = true
   |arrlist.ArrayList::$footprint| = #l2
   |arrlist.ArrayList::$seqLength| = 0

Object #o2/o
   length = 0
   type =java.lang.Object
   exactInstance =false
   |java.lang.Object::<created>| = true
   classInvariant = true
   |arrlist.ArrayList::$footprint| = #l2
   |arrlist.ArrayList::$seqLength| = 0

Object #o3
   length = 0
   type =java.lang.Object
   exactInstance =false
   |java.lang.Object::<created>| = true
   classInvariant = true
   |arrlist.ArrayList::$footprint| = #l2
   |arrlist.ArrayList::$seqLength| = 0

Object #o4
   length = 0
   type =java.lang.Object[]
   exactInstance =true
   |java.lang.Object::<created>| = true
   classInvariant = true
   |arrlist.ArrayList::$footprint| = #l2
   |arrlist.ArrayList::$seqLength| = 0

Object #o5
   length = 0
   type =java.lang.Object
   exactInstance =false
   |java.lang.Object::<created>| = true
   classInvariant = true
   |arrlist.ArrayList::$footprint| = #l2
   |arrlist.ArrayList::$seqLength| = 0
```

```
  Object #o6
    length = 0
    type =arrlist.SelfArrays
    exactInstance =false
    |java.lang.Object::<created>| = true
    classInvariant = true
    |arrlist.ArrayList::$footprint| = #l2
    |arrlist.ArrayList::$seqLength| = 0

  Object #o7
    length = 0
    type =java.lang.Object
    exactInstance =false
    |java.lang.Object::<created>| = true
    classInvariant = true
    |arrlist.ArrayList::$footprint| = #l2
    |arrlist.ArrayList::$seqLength| = 0


Heap anon_1
  Object #o0/null

  Object #o1/self_0
    length = 0
    type =arrlist.ArrayList
    exactInstance =true
    |arrlist.ArrayList::elementData| = #o4
    |arrlist.ArrayList::modCount| = 1
    |arrlist.ArrayList::repr| = #s0
    |arrlist.ArrayList::selfArrays| = #o6
    |arrlist.ArrayList::size| = 0
    |java.lang.Object::<created>| = true
    classInvariant = true
    |arrlist.ArrayList::$footprint| = #l2
    |arrlist.ArrayList::$seqLength| = 0

  Object #o2/o
    length = 0
    type =java.lang.Object
    exactInstance =false
    |java.lang.Object::<created>| = true
    classInvariant = true
    |arrlist.ArrayList::$footprint| = #l2
    |arrlist.ArrayList::$seqLength| = 0

  Object #o3
    length = 0
    type =java.lang.Object
    exactInstance =false
    |java.lang.Object::<created>| = true
    classInvariant = true
    |arrlist.ArrayList::$footprint| = #l2
```

```
      |arrlist.ArrayList::$seqLength| = 0

   Object #o4
      length = 0
      type =java.lang.Object[]
      exactInstance =true
      |java.lang.Object::<created>| = true
      classInvariant = true
      |arrlist.ArrayList::$footprint| = #l2
      |arrlist.ArrayList::$seqLength| = 0

   Object #o5
      length = 0
      type =java.lang.Object
      exactInstance =false
      |java.lang.Object::<created>| = true
      classInvariant = true
      |arrlist.ArrayList::$footprint| = #l2
      |arrlist.ArrayList::$seqLength| = 0

   Object #o6
      length = 0
      type =arrlist.SelfArrays
      exactInstance =false
      |java.lang.Object::<created>| = true
      classInvariant = true
      |arrlist.ArrayList::$footprint| = #l2
      |arrlist.ArrayList::$seqLength| = 0

   Object #o7
      length = 0
      type =java.lang.Object
      exactInstance =false
      |java.lang.Object::<created>| = true
      classInvariant = true
      |arrlist.ArrayList::$footprint| = #l2
      |arrlist.ArrayList::$seqLength| = 0


Heap anon_2
   Object #o0/null

   Object #o1/self_0
      length = 0
      type =arrlist.ArrayList
      exactInstance =true
      |arrlist.ArrayList::elementData| = #o4
      |arrlist.ArrayList::modCount| = 1
      |arrlist.ArrayList::repr| = #s0
      |arrlist.ArrayList::selfArrays| = #o6
      |arrlist.ArrayList::size| = 0
      |java.lang.Object::<created>| = true
```

```
    classInvariant = true
    |arrlist.ArrayList::$footprint| = #l2
    |arrlist.ArrayList::$seqLength| = 0

  Object #o2/o
    length = 0
    type =java.lang.Object
    exactInstance =false
    |java.lang.Object::<created>| = true
    classInvariant = true
    |arrlist.ArrayList::$footprint| = #l2
    |arrlist.ArrayList::$seqLength| = 0

  Object #o3
    length = 0
    type =java.lang.Object
    exactInstance =false
    |java.lang.Object::<created>| = true
    classInvariant = true
    |arrlist.ArrayList::$footprint| = #l2
    |arrlist.ArrayList::$seqLength| = 0

  Object #o4
    length = 0
    type =java.lang.Object[]
    exactInstance =true
    |java.lang.Object::<created>| = true
    classInvariant = true
    |arrlist.ArrayList::$footprint| = #l2
    |arrlist.ArrayList::$seqLength| = 0

  Object #o5
    length = 0
    type =java.lang.Object
    exactInstance =false
    |java.lang.Object::<created>| = true
    classInvariant = true
    |arrlist.ArrayList::$footprint| = #l2
    |arrlist.ArrayList::$seqLength| = 0

  Object #o6
    length = 0
    type =arrlist.SelfArrays
    exactInstance =false
    |java.lang.Object::<created>| = true
    classInvariant = true
    |arrlist.ArrayList::$footprint| = #l2
    |arrlist.ArrayList::$seqLength| = 0

  Object #o7
    length = 0
    type =java.lang.Object
```

```
      exactInstance =false
      |java.lang.Object::<created>| = true
      classInvariant = true
      |arrlist.ArrayList::$footprint| = #l2
      |arrlist.ArrayList::$seqLength| = 0


Heap anon_0
   Object #o0/null

   Object #o1/self_0
      length = 0
      type =arrlist.ArrayList
      exactInstance =true
      |arrlist.ArrayList::elementData| = #o4
      |arrlist.ArrayList::modCount| = 1
      |arrlist.ArrayList::repr| = #s0
      |arrlist.ArrayList::selfArrays| = #o6
      |arrlist.ArrayList::size| = 0
      |java.lang.Object::<created>| = true
      classInvariant = true
      |arrlist.ArrayList::$footprint| = #l2
      |arrlist.ArrayList::$seqLength| = 0

   Object #o2/o
      length = 0
      type =java.lang.Object
      exactInstance =false
      |java.lang.Object::<created>| = true
      classInvariant = true
      |arrlist.ArrayList::$footprint| = #l2
      |arrlist.ArrayList::$seqLength| = 0

   Object #o3
      length = 0
      type =java.lang.Object
      exactInstance =false
      |java.lang.Object::<created>| = true
      classInvariant = true
      |arrlist.ArrayList::$footprint| = #l2
      |arrlist.ArrayList::$seqLength| = 0

   Object #o4
      length = 0
      type =java.lang.Object[]
      exactInstance =true
      |java.lang.Object::<created>| = true
      classInvariant = true
      |arrlist.ArrayList::$footprint| = #l2
      |arrlist.ArrayList::$seqLength| = 0

   Object #o5
```

```
      length = 0
      type =java.lang.Object
      exactInstance =false
      |java.lang.Object::<created>| = true
      classInvariant = true
      |arrlist.ArrayList::$footprint| = #l2
      |arrlist.ArrayList::$seqLength| = 0

   Object #o6
      length = 0
      type =arrlist.SelfArrays
      exactInstance =false
      |java.lang.Object::<created>| = true
      classInvariant = true
      |arrlist.ArrayList::$footprint| = #l2
      |arrlist.ArrayList::$seqLength| = 0

   Object #o7
      length = 0
      type =java.lang.Object
      exactInstance =false
      |java.lang.Object::<created>| = true
      classInvariant = true
      |arrlist.ArrayList::$footprint| = #l2
      |arrlist.ArrayList::$seqLength| = 0



Location Sets
-----------
#l0 = {}
#l1 = {}
#l2 = {(#o1/self_0, |arrlist.ArrayList::repr|), (#o1/self_0,
    |arrlist.ArrayList::size|), (#o1/self_0,
    |arrlist.ArrayList::modCount|), (#o1/self_0,
    |arrlist.ArrayList::elementData|), (#o4, [0]), (#o4, [1]), (#o4, [2]),
    (#o4, [3]), (#o4, |arrlist.ArrayList::repr|), (#o4,
    |arrlist.ArrayList::selfArrays|), (#o4, |arrlist.ArrayList::size|),
    (#o4, |arrlist.ArrayList::modCount|), (#o4,
    |java.lang.Object::<created>|), (#o4,
    |arrlist.ArrayList::elementData|)}
#l3 = {}
#l4 = {(#o0/null, [0]), (#o0/null, [1]), (#o0/null, [2]), (#o0/null,
    [3]), (#o0/null, |arrlist.ArrayList::repr|), (#o0/null,
    |arrlist.ArrayList::selfArrays|), (#o0/null,
    |arrlist.ArrayList::size|), (#o0/null, |arrlist.ArrayList::modCount|),
    (#o0/null, |java.lang.Object::<created>|), (#o0/null,
    |arrlist.ArrayList::elementData|)}
#l5 = {}
#l6 = {}
#l7 = {}
```

```
Sequences
-----------
Seq: #s0/seqEmpty
Length: 0

Seq: #s1
Length: 0

Seq: #s2
Length: 0

Seq: #s3
Length: 0

Seq: #s4
Length: 0

Seq: #s5
Length: 0

Seq: #s6
Length: 0

Seq: #s7
Length: 0
```

C. Anon

C.1. Specified Java Code

```java
public class A2 {

   int x;
   A2 next;

   /*@ requires x == 0;
     @ ensures x == 0;
     @*/
   void m() {
      this.next.next.modx();
   }

   /*@ assignable x;
     @*/
   void modx() {
   }
}
```

C.2. Counterexample for Anon::m

```
Constants
-----------
heap = #h1
|anon.A2::x| = #f8
seqEmpty = #s0
|anon.A2::next| = #f9
|java.lang.Object::<created>| = #f10
anon_heap_modx = #h2
empty = #l1
seqGetOutside = #a28
self = #o1
heapAfter_modx = #h0
exc_0 = #o0
null = #o0


Heaps
-----------
Heap heap
   Object #o0/exc_0/null

   Object #o1/self
      length = 0
      type =anon.A2
      exactInstance =true
      |anon.A2::next| = #o4
      |anon.A2::x| = 0
      |java.lang.Object::<created>| = true
      classInvariant = true

   Object #o2
      length = 0
      type =anon.A2
      exactInstance =false
      |anon.A2::next| = #o0/exc_0/null
      |anon.A2::x| = 0
      |java.lang.Object::<created>| = true
      classInvariant = false

   Object #o3
      length = 0
      type =anon.A2
      exactInstance =false
      |anon.A2::next| = #o0/exc_0/null
      |anon.A2::x| = 0
      |java.lang.Object::<created>| = false
      classInvariant = false

   Object #o4
      length = 0
      type =anon.A2
```

```
      exactInstance =false
      |anon.A2::next| = #o1/self
      |anon.A2::x| = 0
      |java.lang.Object::<created>| = true
      classInvariant = false

   Object #o5
      length = 0
      type =java.lang.Object
      exactInstance =false
      |java.lang.Object::<created>| = false
      classInvariant = false

   Object #o6
      length = 0
      type =java.lang.Object
      exactInstance =false
      |java.lang.Object::<created>| = false
      classInvariant = false

   Object #o7
      length = 0
      type =java.lang.Object
      exactInstance =false
      |java.lang.Object::<created>| = false
      classInvariant = false


Heap heapAfter_modx
   Object #o0/exc_0/null

   Object #o1/self
      length = 0
      type =anon.A2
      exactInstance =true
      |anon.A2::next| = #o4
      |anon.A2::x| = 2
      |java.lang.Object::<created>| = true
      classInvariant = true

   Object #o2
      length = 0
      type =anon.A2
      exactInstance =false
      |anon.A2::next| = #o0/exc_0/null
      |anon.A2::x| = 0
      |java.lang.Object::<created>| = true
      classInvariant = false

   Object #o3
      length = 0
      type =anon.A2
```

```
    exactInstance =false
    |anon.A2::next| = #o0/exc_0/null
    |anon.A2::x| = 2
    |java.lang.Object::<created>| = false
    classInvariant = false

  Object #o4
    length = 0
    type =anon.A2
    exactInstance =false
    |anon.A2::next| = #o1/self
    |anon.A2::x| = 0
    |java.lang.Object::<created>| = true
    classInvariant = false

  Object #o5
    length = 0
    type =java.lang.Object
    exactInstance =false
    |java.lang.Object::<created>| = true
    classInvariant = false

  Object #o6
    length = 0
    type =java.lang.Object
    exactInstance =false
    |java.lang.Object::<created>| = true
    classInvariant = false

  Object #o7
    length = 0
    type =java.lang.Object
    exactInstance =false
    |java.lang.Object::<created>| = true
    classInvariant = false


Heap anon_heap_modx
  Object #o0/exc_0/null

  Object #o1/self
    length = 0
    type =anon.A2
    exactInstance =true
    |anon.A2::next| = #o0/exc_0/null
    |anon.A2::x| = 2
    |java.lang.Object::<created>| = false
    classInvariant = false

  Object #o2
    length = 0
    type =anon.A2
```

```
    exactInstance =false
    |anon.A2::next| = #o2
    |anon.A2::x| = 0
    |java.lang.Object::<created>| = true
    classInvariant = false

  Object #o3
    length = 0
    type =anon.A2
    exactInstance =false
    |anon.A2::next| = #o0/exc_0/null
    |anon.A2::x| = 2
    |java.lang.Object::<created>| = false
    classInvariant = false

  Object #o4
    length = 0
    type =anon.A2
    exactInstance =false
    |anon.A2::next| = #o0/exc_0/null
    |anon.A2::x| = 0
    |java.lang.Object::<created>| = true
    classInvariant = false

  Object #o5
    length = 0
    type =java.lang.Object
    exactInstance =false
    |java.lang.Object::<created>| = true
    classInvariant = false

  Object #o6
    length = 0
    type =java.lang.Object
    exactInstance =false
    |java.lang.Object::<created>| = true
    classInvariant = false

  Object #o7
    length = 0
    type =java.lang.Object
    exactInstance =false
    |java.lang.Object::<created>| = true
    classInvariant = false



Location Sets
-----------
#l0 = {}
#l1 = {}
#l2 = {}
```

```
#13 = {}
#14 = {}
#15 = {}
#16 = {}
#17 = {}

Sequences
-----------
Seq: #s0/seqEmpty
Length: 0

Seq: #s1
Length: 0

Seq: #s2
Length: 0

Seq: #s3
Length: 0

Seq: #s4
Length: 0

Seq: #s5
Length: 0

Seq: #s6
Length: 0

Seq: #s7
Length: 0
```

D. Cell

D.1. Specified Java Code

```
class Cell {
    private int x;
    private int y;


    /*@ normal_behavior
      @ assignable \nothing;
      @ ensures getX() == 0;
      @ ensures \fresh(footprint);
      @*/
    Cell() {
    }


    /*@ normal_behavior
```

```
      @ assignable \nothing;
      @ accessible footprint;
      @ ensures \result == getX();
      @*/
    int getX() {
        return x;
    }


    /*@ normal_behavior
      @ assignable footprint;
      @ ensures getX() == value;
      @ ensures \new_elems_fresh(footprint);
      @*/
    void setX(int value) {
        x = value;
    }

    /*@ model \locset footprint;
      @ accessible footprint: footprint;
      @ represents footprint = y;
      @*/

    //@ accessible \inv: \nothing;
}
```

D.2. Counterexample for Cell::setX

```

Constants
-----------
store_1 = #h2
heap = #h0
store_0 = #h1
getX_sk_0 = 0
seqEmpty = #s0
|cell.Cell::x| = #f9
|cell.Cell::y| = #f8
|java.lang.Object::<created>| = #f10
value = 0
empty = #l2
seqGetOutside = #a41
self = #o1
null = #o0


Heaps
-----------
Heap heap
   Object #o0/null

```

```
Object #o1/self
   length = 0
   type =cell.Cell
   exactInstance =true
   |cell.Cell::x| = -4
   |cell.Cell::y| = -4
   |java.lang.Object::<created>| = true
   |cell.Cell::$footprint| = #l0
   classInvariant = true

Object #o2
   length = 0
   type =java.lang.Object
   exactInstance =false
   |java.lang.Object::<created>| = false
   |cell.Cell::$footprint| = #l0
   classInvariant = true

Object #o3
   length = 0
   type =java.lang.Object
   exactInstance =false
   |java.lang.Object::<created>| = false
   |cell.Cell::$footprint| = #l0
   classInvariant = true

Object #o4
   length = 0
   type =java.lang.Object
   exactInstance =false
   |java.lang.Object::<created>| = false
   |cell.Cell::$footprint| = #l0
   classInvariant = true

Object #o5
   length = 0
   type =java.lang.Object
   exactInstance =false
   |java.lang.Object::<created>| = false
   |cell.Cell::$footprint| = #l0
   classInvariant = true

Object #o6
   length = 0
   type =java.lang.Object
   exactInstance =false
   |java.lang.Object::<created>| = false
   |cell.Cell::$footprint| = #l0
   classInvariant = true

Object #o7
   length = 0
```

```
      type =java.lang.Object
      exactInstance =false
      |java.lang.Object::<created>| = false
      |cell.Cell::$footprint| = #l0
      classInvariant = true


Heap store_1
   Object #o0/null

   Object #o1/self
      length = 0
      type =cell.Cell
      exactInstance =true
      |cell.Cell::x| = 0
      |cell.Cell::y| = -4
      |java.lang.Object::<created>| = true
      |cell.Cell::$footprint| = #l0
      classInvariant = true

   Object #o2
      length = 0
      type =java.lang.Object
      exactInstance =false
      |java.lang.Object::<created>| = false
      |cell.Cell::$footprint| = #l0
      classInvariant = true

   Object #o3
      length = 0
      type =java.lang.Object
      exactInstance =false
      |java.lang.Object::<created>| = false
      |cell.Cell::$footprint| = #l0
      classInvariant = true

   Object #o4
      length = 0
      type =java.lang.Object
      exactInstance =false
      |java.lang.Object::<created>| = false
      |cell.Cell::$footprint| = #l0
      classInvariant = true

   Object #o5
      length = 0
      type =java.lang.Object
      exactInstance =false
      |java.lang.Object::<created>| = false
      |cell.Cell::$footprint| = #l0
      classInvariant = true

```

```
  Object #o6
    length = 0
    type =java.lang.Object
    exactInstance =false
    |java.lang.Object::<created>| = false
    |cell.Cell::$footprint| = #l0
    classInvariant = true

  Object #o7
    length = 0
    type =java.lang.Object
    exactInstance =false
    |java.lang.Object::<created>| = false
    |cell.Cell::$footprint| = #l0
    classInvariant = true


Heap store_0
  Object #o0/null

  Object #o1/self
    length = 0
    type =cell.Cell
    exactInstance =true
    |cell.Cell::x| = 0
    |cell.Cell::y| = -4
    |java.lang.Object::<created>| = true
    |cell.Cell::$footprint| = #l0
    classInvariant = true

  Object #o2
    length = 0
    type =java.lang.Object
    exactInstance =false
    |java.lang.Object::<created>| = false
    |cell.Cell::$footprint| = #l0
    classInvariant = true

  Object #o3
    length = 0
    type =java.lang.Object
    exactInstance =false
    |java.lang.Object::<created>| = false
    |cell.Cell::$footprint| = #l0
    classInvariant = true

  Object #o4
    length = 0
    type =java.lang.Object
    exactInstance =false
    |java.lang.Object::<created>| = false
    |cell.Cell::$footprint| = #l0
```

```
      classInvariant = true

   Object #o5
      length = 0
      type =java.lang.Object
      exactInstance =false
      |java.lang.Object::<created>| = false
      |cell.Cell::$footprint| = #l0
      classInvariant = true

   Object #o6
      length = 0
      type =java.lang.Object
      exactInstance =false
      |java.lang.Object::<created>| = false
      |cell.Cell::$footprint| = #l0
      classInvariant = true

   Object #o7
      length = 0
      type =java.lang.Object
      exactInstance =false
      |java.lang.Object::<created>| = false
      |cell.Cell::$footprint| = #l0
      classInvariant = true



Location Sets
-----------
#l0 = {(#o1/self, |cell.Cell::y|)}
#l1 = {(#o0/null, |cell.Cell::y|)}
#l2 = {}
#l3 = {}
#l4 = {}
#l5 = {}
#l6 = {}
#l7 = {}

Sequences
-----------
Seq: #s0/seqEmpty
Length: 0

Seq: #s1
Length: 0

Seq: #s2
Length: 0

Seq: #s3
Length: 0
```

```

Seq: #s4
Length: 0

Seq: #s5
Length: 0

Seq: #s6
Length: 0

Seq: #s7
Length: 0
```

E. SimplifiedLL

E.1. Specified Java Code

```java
final class SimplifiedLinkedList {

    private /*@nullable@*/ Node first;
    private int size;

    /*@ private ghost \seq nodeseq; */

    /*@
      @ private invariant (\forall int i; 0<=i && i<size;
      @        ((Node)nodeseq[i]) != null // this implies \typeof(nodeseq[i]) == \type(Node)
      @     && (\forall int j; 0<=j && j<size; (Node)nodeseq[i] == (Node)nodeseq[j] ==> i == j)
      @     && ((Node)nodeseq[i]).next == (i==size-1 ? null : (Node)nodeseq[i+1]));
      @
      @ private invariant size > 0;
      @ private invariant first == (Node)nodeseq[0];
      @*/


    /*@ normal_behaviour
      @ requires n >= 0 && n < size && \invariant_for(this);
      @ ensures \result == (Node)nodeseq[n];
      @ assignable \strictly_nothing;
      @ helper */
    private Node getNext(int n) {
        Node result = first;
        /*@ loop_invariant
          @   0<=i && i <=n && result == (Node)nodeseq[i];
          @ decreases n-i;
          @ assignable \strictly_nothing;
          @*/
        for(int i = 0; i < n; i++) {
```

```
            result = result.next;
        }
        return result;
    }

    /*@ normal_behaviour
      @ requires i > 0 && i < size;
      @ ensures nodeseq == \old(\seq_concat(nodeseq[0..i-1],
        nodeseq[i+1..nodeseq.length-1]));
      @*/
    public void remove(int i) {
        Node node = getNext(i-1);
        Node node2 = getNext(i);
        node.next = node2.next;
        //@ set nodeseq = (\seq_concat(\seq_sub(nodeseq,0,i-1),
            \seq_sub(nodeseq,i+1,\seq_length(nodeseq)-1)));
        size --;
    }
}

final class Node {
    public /*@nullable@*/ Node next;
    public Object data;
}
```

E.2. Counterexample for SimplifiedLL.remove

```

Constants
-----------
heap = #h0
seqEmpty = #s0
|java.lang.Object::<created>| = #f8
empty = #l6
value = 0
seqGetOutside = #a24
self = #o4
null = #o0
array = #o1


Heaps
-----------
Heap heap
   Object #o0/null

   Object #o1/array
      length = 1
      type =int[][]
      exactInstance =true
      |java.lang.Object::<created>| = true
```

```
      classInvariant = true
      [0] = #o2

   Object #o2
      length = 3
      type =int[]
      exactInstance =true
      |java.lang.Object::<created>| = true
      classInvariant = true
      [0] = 1
      [1] = 1
      [2] = 2

   Object #o3
      length = 3
      type =java.lang.Object
      exactInstance =false
      |java.lang.Object::<created>| = false
      classInvariant = true

   Object #o4/self
      length = 3
      type =saddleback.Saddleback
      exactInstance =true
      |java.lang.Object::<created>| = true
      classInvariant = true

   Object #o5
      length = 3
      type =java.lang.Cloneable
      exactInstance =false
      |java.lang.Object::<created>| = false
      classInvariant = true

   Object #o6
      length = 3
      type =java.lang.Object
      exactInstance =false
      |java.lang.Object::<created>| = false
      classInvariant = true

   Object #o7
      length = 3
      type =java.lang.Cloneable
      exactInstance =false
      |java.lang.Object::<created>| = false
      classInvariant = true



Location Sets
-----------
```

```
#l0 = {}
#l1 = {}
#l2 = {}
#l3 = {}
#l4 = {}
#l5 = {}
#l6 = {}
#l7 = {}

Sequences
-----------
Seq: #s0/seqEmpty
Length: 0

Seq: #s1
Length: 0

Seq: #s2
Length: 0

Seq: #s3
Length: 0

Seq: #s4
Length: 0

Seq: #s5
Length: 0

Seq: #s6
Length: 0

Seq: #s7
Length: 0
```

F. SaddleBack

F.1. Specified Java Code

```
class Saddleback {

    /*@ public normal_behaviour
      @ requires (\forall int i; 0<=i && i<array.length;
      @     array[i].length == array[0].length);
      @
      @ requires array.length > 0;
      @ requires array[0].length > 0;
      @
      @ requires (\forall int k,i,j;
      @    0<=k && k < i && i < array.length && 0<=j && j <
          array[0].length;
```

```
  @      array[k][j] <= array[i][j]);
  @
  @ requires (\forall int k,j,i;
  @    0<=i && i < array.length && 0<=k && k<j && j < array[i].length;
  @      array[i][k] <= array[i][j]);
  @
  @ ensures \result == null ==>
  @     (\forall int i; 0<=i && i<array.length;
  @       (\forall int j; 0<=j && j<array[i].length;
  @         array[i][j] != value));
  @
  @ ensures \result != null ==>
  @     \result.length == 2 &&
  @     array[\result[0]][\result[1]] == value;
  @
  @ modifies \nothing;
  @*/
public /*@nullable*/ int[] search(int[][] array, int value) {
    int x = 0;
    int y = array[0].length - 1;

    /*@
      @ loop_invariant
      @   0 <= x && x <= array.length &&
      @  -1 <= y && y < array[0].length &&
      @   (\forall int j,i; 0<=i && i < array.length &&
      @                     0<=j && j < array[0].length ;
      @      (i < x || j > y) ==> array[i][j] != value);
      @
      @ decreases array.length - x - y;
      @ modifies \nothing;
      @*/
    while(x < array.length && y >= 0) {

         if(array[x][y] == value) {
                    return new int[] { x, y };
        }

         if(array[x][y] < value) {
            x++;
         } else {
            y--;
         }

    }

    return null;
  }
}
```

F.2. Counterexample for Saddleback::search

```
Constants
-----------
heap = #h0
seqEmpty = #s0
|java.lang.Object::<created>| = #f8
empty = #l6
value = 0
seqGetOutside = #a24
self = #o4
null = #o0
array = #o1


Heaps
-----------
Heap heap
   Object #o0/null

   Object #o1/array
      length = 1
      type =int[][]
      exactInstance =true
      |java.lang.Object::<created>| = true
      classInvariant = true
      [0] = #o2

   Object #o2
      length = 3
      type =int[]
      exactInstance =true
      |java.lang.Object::<created>| = true
      classInvariant = true
      [0] = 1
      [1] = 1
      [2] = 2

   Object #o3
      length = 3
      type =java.lang.Object
      exactInstance =false
      |java.lang.Object::<created>| = false
      classInvariant = true

   Object #o4/self
      length = 3
      type =saddleback.Saddleback
      exactInstance =true
      |java.lang.Object::<created>| = true
      classInvariant = true

   Object #o5
```

```
      length = 3
      type =java.lang.Cloneable
      exactInstance =false
      |java.lang.Object::<created>| = false
      classInvariant = true

   Object #o6
      length = 3
      type =java.lang.Object
      exactInstance =false
      |java.lang.Object::<created>| = false
      classInvariant = true

   Object #o7
      length = 3
      type =java.lang.Cloneable
      exactInstance =false
      |java.lang.Object::<created>| = false
      classInvariant = true



Location Sets
-----------
#l0 = {}
#l1 = {}
#l2 = {}
#l3 = {}
#l4 = {}
#l5 = {}
#l6 = {}
#l7 = {}

Sequences
-----------
Seq: #s0/seqEmpty
Length: 0

Seq: #s1
Length: 0

Seq: #s2
Length: 0

Seq: #s3
Length: 0

Seq: #s4
Length: 0

Seq: #s5
Length: 0
```

```

Seq: #s6
Length: 0

Seq: #s7
Length: 0
```

G. RingBuffer

G.1. Specified Java Code

```java
public class RingBuffer {

    int[] data;
    int first;
    int len;

    //@ ghost \seq list;
    //@ invariant list.length == len;
    //@ invariant data.length > 0;
    //@ invariant 0 <= first && first < data.length;
    //@ invariant 0 <= len && len <= data.length;
    //@ invariant (\forall int i; 0 <= i && i < len; list[i] ==
        data[modulo(first+i)]);

    /*@ normal_behavior
      @ requires n >= 1;
      @ assignable this.*;
      @ ensures list == \seq_empty;
      @ ensures first == 0;
      @ ensures data.length == n;
     */
    RingBuffer(int n){
        //@ set list = \seq_empty;
        data = new int[n];
    }

    /*@ normal_behavior
      @ ensures list == \seq_empty;
      @ ensures first == \old(first);
      @ assignable len,list;
      @*/
    void clear(){
        //@ set list = \seq_empty;
        len = 0;
    }

    /*@ normal_behavior
      @ requires !isEmpty();
      @ ensures \result == list[0];
      @ pure
```

```
      @*/
    int head() {
        return data[first];
    }

    /*@ normal_behavior
      @ requires !isFull();
      @ ensures list == \seq_concat(\old(list),\seq_singleton(x));
      @ assignable len,list,data[modulo(first+len)];
      @*/
    void push(int x){
        int pos = modulo(first+len);

        data[pos] = x;
        //@ set list = \seq_concat(list,\seq_singleton(x));
        len = len + 2;
    }

    /*@ normal_behavior
      @ requires !isEmpty();
      @ ensures list == \seq_sub(\old(list),1,\old(list.length)-1);
      @ ensures first == modulo(\old(first)+1);
      @ ensures \result == \old(data[first]);
      @ assignable first,len,list;
      @*/
    int pop(){
        int r = data[first];
        first = modulo(first+1);

        len = len -2;
        //@ set list = \seq_sub(list,1,\seq_length(list)-1);
        return r;
    }


    // helper methods
    /*@ normal_behavior
      @ ensures \result == (len == 0);
      @ pure helper
      @*/
    boolean isEmpty() {
        return len == 0;
    }

    /*@ normal_behavior
      @ ensures \result == (len == data.length);
      @ pure
      @*/
    boolean isFull() {
        return len == data.length;
    }

```

```
    /*@ public normal_behaviour
      @   ensures x >= 0 && x < data.length ==> \result == x;
      @   ensures x >= data.length && x < data.length + data.length ==>
      @       \result == x - data.length;
      @   pure
      @*/
    int modulo(int x) {
        return x < data.length ? x : x - data.length;
    }

    // Harness

    //@ ensures true;
    //@ signals (Exception) false;
    static void test (int x, int y, int z){
        RingBuffer b = new RingBuffer(2);
        b.push(x);
        b.push(y);
        int h = b.pop();
        assert h == x;
        b.push(z);
        h = b.pop();
        assert h == y;
        h = b.pop();
        assert h == z;
    }

}
```

G.2. Counterexample for RingBuffer::push

```
Constants
-----------
seqEmpty = #s0
result = 0
|java.lang.Object::<created>| = #f12
|ringbuffer.RingBuffer::first| = #f8
self = #o1
|ringbuffer.RingBuffer::len| = #f9
anon_heap_modulo = #h2
heap = #h0
seqSingleton_4 = #s1
store_0 = #h4
seqSingleton_3 = #s2
heapAfter_modulo = #h0
empty = #l1
|ringbuffer.RingBuffer::list| = #f10
null = #o0
seqConcat_4 = #s1
|ringbuffer.RingBuffer::data| = #f11
seqSingleton_1 = #s2
```

```
seqConcat_1 = #s1
seqSingleton_2 = #s4
exc_0 = #o0
seqConcat_0 = #s1
seqConcat_3 = #s4
seqConcat_2 = #s4
seqSingleton_0 = #s2
modulo_sk_1 = 0
isFull_sk_3 = false
seqGetOutside = #a0
x = 0


Heaps
-----------
Heap anon_heap_modulo
   Object #o0/null/exc_0

   Object #o1/self
      length = 1
      type =ringbuffer.RingBuffer
      exactInstance =true
      |java.lang.Object::<created>| = false
      |ringbuffer.RingBuffer::data| = #o0/null/exc_0
      |ringbuffer.RingBuffer::first| = -4
      |ringbuffer.RingBuffer::len| = 2
      |ringbuffer.RingBuffer::list| = #s4
      classInvariant = false

   Object #o2
      length = 1
      type =java.util.List
      exactInstance =false
      |java.lang.Object::<created>| = false
      classInvariant = false

   Object #o3
      length = 1
      type =java.util.List
      exactInstance =false
      |java.lang.Object::<created>| = false
      classInvariant = false

   Object #o4
      length = 1
      type =int[]
      exactInstance =true
      |java.lang.Object::<created>| = false
      classInvariant = false
      [0] = 0

   Object #o5
```

```
      length = 1
      type =java.util.List
      exactInstance =false
      |java.lang.Object::<created>| = false
      classInvariant = false

   Object #o6
      length = 1
      type =java.util.List
      exactInstance =false
      |java.lang.Object::<created>| = false
      classInvariant = false

   Object #o7
      length = 1
      type =java.util.List
      exactInstance =false
      |java.lang.Object::<created>| = false
      classInvariant = false


Heap heap
   Object #o0/null/exc_0

   Object #o1/self
      length = 1
      type =ringbuffer.RingBuffer
      exactInstance =true
      |java.lang.Object::<created>| = true
      |ringbuffer.RingBuffer::data| = #o4
      |ringbuffer.RingBuffer::first| = 0
      |ringbuffer.RingBuffer::len| = 0
      |ringbuffer.RingBuffer::list| = #s0
      classInvariant = true

   Object #o2
      length = 1
      type =java.util.List
      exactInstance =false
      |java.lang.Object::<created>| = true
      classInvariant = false

   Object #o3
      length = 1
      type =java.util.List
      exactInstance =false
      |java.lang.Object::<created>| = true
      classInvariant = false

   Object #o4
      length = 1
      type =int[]
```

```
      exactInstance =true
      |java.lang.Object::<created>| = true
      classInvariant = false
      [0] = 0

   Object #o5
      length = 1
      type =java.util.List
      exactInstance =false
      |java.lang.Object::<created>| = true
      classInvariant = false

   Object #o6
      length = 1
      type =java.util.List
      exactInstance =false
      |java.lang.Object::<created>| = true
      classInvariant = false

   Object #o7
      length = 1
      type =java.util.List
      exactInstance =false
      |java.lang.Object::<created>| = true
      classInvariant = false


Heap store_0
   Object #o0/null/exc_0

   Object #o1/self
      length = 1
      type =ringbuffer.RingBuffer
      exactInstance =true
      |java.lang.Object::<created>| = true
      |ringbuffer.RingBuffer::data| = #o4
      |ringbuffer.RingBuffer::first| = 0
      |ringbuffer.RingBuffer::len| = 2
      |ringbuffer.RingBuffer::list| = #s1
      classInvariant = false

   Object #o2
      length = 1
      type =java.util.List
      exactInstance =false
      |java.lang.Object::<created>| = true
      classInvariant = false

   Object #o3
      length = 1
      type =java.util.List
      exactInstance =false
```

```
      |java.lang.Object::<created>| = true
      classInvariant = false

   Object #o4
      length = 1
      type =int[]
      exactInstance =true
      |java.lang.Object::<created>| = true
      classInvariant = false
      [0] = 0

   Object #o5
      length = 1
      type =java.util.List
      exactInstance =false
      |java.lang.Object::<created>| = true
      classInvariant = false

   Object #o6
      length = 1
      type =java.util.List
      exactInstance =false
      |java.lang.Object::<created>| = true
      classInvariant = false

   Object #o7
      length = 1
      type =java.util.List
      exactInstance =false
      |java.lang.Object::<created>| = true
      classInvariant = false


Heap heapAfter_modulo
   Object #o0/null/exc_0

   Object #o1/self
      length = 1
      type =ringbuffer.RingBuffer
      exactInstance =true
      |java.lang.Object::<created>| = true
      |ringbuffer.RingBuffer::data| = #o4
      |ringbuffer.RingBuffer::first| = 0
      |ringbuffer.RingBuffer::len| = 0
      |ringbuffer.RingBuffer::list| = #s0
      classInvariant = true

   Object #o2
      length = 1
      type =java.util.List
      exactInstance =false
      |java.lang.Object::<created>| = true
```

```
    classInvariant = false

  Object #o3
    length = 1
    type =java.util.List
    exactInstance =false
    |java.lang.Object::<created>| = true
    classInvariant = false

  Object #o4
    length = 1
    type =int[]
    exactInstance =true
    |java.lang.Object::<created>| = true
    classInvariant = false
    [0] = 0

  Object #o5
    length = 1
    type =java.util.List
    exactInstance =false
    |java.lang.Object::<created>| = true
    classInvariant = false

  Object #o6
    length = 1
    type =java.util.List
    exactInstance =false
    |java.lang.Object::<created>| = true
    classInvariant = false

  Object #o7
    length = 1
    type =java.util.List
    exactInstance =false
    |java.lang.Object::<created>| = true
    classInvariant = false



Location Sets
-----------
#l0 = {}
#l1 = {}
#l2 = {}
#l3 = {}
#l4 = {}
#l5 = {}
#l6 = {}
#l7 = {}

Sequences
```

```
-----------
Seq: #s0/seqEmpty
Length: 0

Seq: #s1/seqSingleton_4/seqConcat_4/seqConcat_1/seqConcat_0
Length: 1
[0] = 0

Seq: #s2/seqSingleton_3/seqSingleton_1/seqSingleton_0
Length: 1
[0] = 0

Seq: #s3
Length: 0

Seq: #s4/seqSingleton_2/seqConcat_3/seqConcat_2
Length: 1
[0] = 0

Seq: #s5
Length: 0

Seq: #s6
Length: 0

Seq: #s7
Length: 0
```

G.3. Counterexample for RingBuffer::pop

```
Constants
-----------
seqEmpty = #s0
isEmpty_sk_3 = false
|java.lang.Object::<created>| = #f12
|ringbuffer.RingBuffer::first| = #f8
|ringbuffer.RingBuffer::data| = #f11
result_0 = 0
self = #o7
|ringbuffer.RingBuffer::len| = #f9
exc_0 = #o0
anon_heap_modulo = #h0
store_1 = #h0
heap = #h1
modulo_sk_11 = 0
heapAfter_modulo = #h2
seqSub_8 = #s0
seqSub_9 = #s0
seqSub_6 = #s4
seqSub_7 = #s1
empty = #l6
```

```
seqGetOutside = #a0
seqSub_5 = #s0
|ringbuffer.RingBuffer::list| = #f10
null = #o0


Heaps
-----------
Heap anon_heap_modulo
   Object #o0/exc_0/null

   Object #o1
      length = 0
      type =java.util.Set
      exactInstance =false
      |java.lang.Object::<created>| = false
      classInvariant = false

   Object #o2
      length = 0
      type =java.util.Set
      exactInstance =false
      |java.lang.Object::<created>| = false
      classInvariant = false

   Object #o3
      length = 0
      type =java.util.Set
      exactInstance =false
      |java.lang.Object::<created>| = false
      classInvariant = false

   Object #o4
      length = 1
      type =int[]
      exactInstance =true
      |java.lang.Object::<created>| = true
      classInvariant = false
      [0] = 2

   Object #o5
      length = 0
      type =java.util.Set
      exactInstance =false
      |java.lang.Object::<created>| = false
      classInvariant = false

   Object #o6
      length = 0
      type =java.util.Set
      exactInstance =false
      |java.lang.Object::<created>| = false
```

```
      classInvariant = false

   Object #o7/self
      length = 0
      type =ringbuffer.RingBuffer
      exactInstance =true
      |java.lang.Object::<created>| = true
      |ringbuffer.RingBuffer::data| = #o4
      |ringbuffer.RingBuffer::first| = 0
      |ringbuffer.RingBuffer::len| = -1
      |ringbuffer.RingBuffer::list| = #s4
      classInvariant = false


Heap heap
   Object #o0/exc_0/null

   Object #o1
      length = 0
      type =java.util.Set
      exactInstance =false
      |java.lang.Object::<created>| = false
      classInvariant = false

   Object #o2
      length = 0
      type =java.util.Set
      exactInstance =false
      |java.lang.Object::<created>| = false
      classInvariant = false

   Object #o3
      length = 0
      type =java.util.Set
      exactInstance =false
      |java.lang.Object::<created>| = false
      classInvariant = false

   Object #o4
      length = 1
      type =int[]
      exactInstance =true
      |java.lang.Object::<created>| = true
      classInvariant = false
      [0] = 2

   Object #o5
      length = 0
      type =java.util.Set
      exactInstance =false
      |java.lang.Object::<created>| = false
      classInvariant = false
```

```

  Object #o6
    length = 0
    type =java.util.Set
    exactInstance =false
    |java.lang.Object::<created>| = false
    classInvariant = false

  Object #o7/self
    length = 0
    type =ringbuffer.RingBuffer
    exactInstance =true
    |java.lang.Object::<created>| = true
    |ringbuffer.RingBuffer::data| = #o4
    |ringbuffer.RingBuffer::first| = 0
    |ringbuffer.RingBuffer::len| = 1
    |ringbuffer.RingBuffer::list| = #s2
    classInvariant = true


Heap store_1
  Object #o0/exc_0/null

  Object #o1
    length = 0
    type =java.util.Set
    exactInstance =false
    |java.lang.Object::<created>| = false
    classInvariant = false

  Object #o2
    length = 0
    type =java.util.Set
    exactInstance =false
    |java.lang.Object::<created>| = false
    classInvariant = false

  Object #o3
    length = 0
    type =java.util.Set
    exactInstance =false
    |java.lang.Object::<created>| = false
    classInvariant = false

  Object #o4
    length = 1
    type =int[]
    exactInstance =true
    |java.lang.Object::<created>| = true
    classInvariant = false
    [0] = 2

```

```
  Object #o5
    length = 0
    type =java.util.Set
    exactInstance =false
    |java.lang.Object::<created>| = false
    classInvariant = false

  Object #o6
    length = 0
    type =java.util.Set
    exactInstance =false
    |java.lang.Object::<created>| = false
    classInvariant = false

  Object #o7/self
    length = 0
    type =ringbuffer.RingBuffer
    exactInstance =true
    |java.lang.Object::<created>| = true
    |ringbuffer.RingBuffer::data| = #o4
    |ringbuffer.RingBuffer::first| = 0
    |ringbuffer.RingBuffer::len| = -1
    |ringbuffer.RingBuffer::list| = #s4
    classInvariant = false


Heap heapAfter_modulo
  Object #o0/exc_0/null

  Object #o1
    length = 0
    type =java.util.Set
    exactInstance =false
    |java.lang.Object::<created>| = false
    classInvariant = false

  Object #o2
    length = 0
    type =java.util.Set
    exactInstance =false
    |java.lang.Object::<created>| = false
    classInvariant = false

  Object #o3
    length = 0
    type =java.util.Set
    exactInstance =false
    |java.lang.Object::<created>| = false
    classInvariant = false

  Object #o4
    length = 1
```

```
      type =int[]
      exactInstance =true
      |java.lang.Object::<created>| = true
      classInvariant = false
      [0] = 2

   Object #o5
      length = 0
      type =java.util.Set
      exactInstance =false
      |java.lang.Object::<created>| = false
      classInvariant = false

   Object #o6
      length = 0
      type =java.util.Set
      exactInstance =false
      |java.lang.Object::<created>| = false
      classInvariant = false

   Object #o7/self
      length = 0
      type =ringbuffer.RingBuffer
      exactInstance =true
      |java.lang.Object::<created>| = true
      |ringbuffer.RingBuffer::data| = #o4
      |ringbuffer.RingBuffer::first| = 0
      |ringbuffer.RingBuffer::len| = 1
      |ringbuffer.RingBuffer::list| = #s2
      classInvariant = true



Location Sets
-----------
#l0 = {}
#l1 = {}
#l2 = {}
#l3 = {}
#l4 = {}
#l5 = {}
#l6 = {}
#l7 = {}

Sequences
-----------
Seq: #s0/seqEmpty/seqSub_8/seqSub_9/seqSub_5
Length: 0

Seq: #s1/seqSub_7
Length: 0
```

```
Seq: #s2
Length: 1
[0] = 2

Seq: #s3
Length: 0

Seq: #s4/seqSub_6
Length: 0

Seq: #s5
Length: 0

Seq: #s6
Length: 0

Seq: #s7
Length: 0
```