Specification Model Library for the Interactive Program Prover JIVE

Marcello Miragliotta

23.09.2004

Abstract

This paper deals with the transformation of JML model classes to abstract data type definitions that can be used in a theorem prover like Isabelle/HOL. It will show some different approaches that can be used to achieve this transformation, such like primitive lists or inductively defined sets for some model classes. The two model classes JMLMath and JMLObjectSequence will help us to cover as much aspects as possible of JML. This becomes clear when we look at the JML specification of the two classes: JMLMath actually consists only of static declarations whereas, JMLObjectSequence will cover quite a lot of the dynamic aspects of the language. The main goal is to develop some ideas concerning automated transformation and to discuss them.

Contents

1 Introduction **5**
1.1 Motivation . . . 5
1.2 Aims . . . 5
1.3 JML - Java Modeling Language . . . 5
1.3.1 The Specification Language . . . 5
1.3.2 JML Model Classes . . . 8
1.4 Isabelle/HOL . . . 8
1.4.1 Theories . . . 9
1.4.2 A Little Theory Example . . . 9
1.5 General Restrictions of the Modelling . . . 10

2 Model Class JMLMath **11**
2.1 General Remarks . . . 11
2.2 Predefined Isabelle Theories . . . 11
2.3 Transformation . . . 11
2.3.1 Inheritance . . . 11
2.3.2 Fields . . . 11
2.3.3 Invariants . . . 11
2.3.4 Methods . . . 11
2.3.5 Pre- and Post-Conditions . . . 13
2.4 Summary . . . 16

3 Model Class JMLObjectSequence **17**
3.1 General Remarks . . . 17
3.2 Predefined Isabelle Theories . . . 17
3.3 First Approach: Modelling the Sequence as Primitive List . . . 17
3.3.1 Overview . . . 17
3.3.2 Restrictions . . . 18
3.3.3 Definitions . . . 18
3.3.4 Pre- and Post-Conditions . . . 21
3.3.5 Equational Theory . . . 24
3.3.6 Summary . . . 24
3.4 Second Approach: Inductively Defined Sets . . . 24
3.4.1 Overview . . . 24
3.4.2 Restrictions . . . 25
3.4.3 Objects, Classes and Inductively Defined Sets . . . 25
3.4.4 Model and Ghost Fields . . . 26
3.4.5 Invariants . . . 26
3.4.6 Summary . . . 26

4 Automated Transformation of JML model classes **27**
4.1 General Remarks . . . 27
4.2 A Generic Model . . . 27
4.2.1 Axiomatic Specification . . . 27
4.2.2 The Refinement of a Model Class . . . 28
4.3 Automated Generation Algorithm . . . 28
4.4 Summary . . . 30

5 Conclusions **32**
5.1 Achievements . . . 32
5.2 Further Work . . . 32
5.3 Final Statement . . . 32

References **33**

A Figures **34**

B Tables **34**

C Isabelle Theories **34**
C.1 JMLMath 34
C.2 JMLObjectSequence - Approach Primitive Lists 35
C.3 JMLObjectSequence - Approach Inductive Sets 45

1 Introduction

1.1 Motivation

In a co-operative effort, the Software Component Technology Group together with the Softwaretechnik Group at TU Kaiserslautern works on the development of an interactive program prover, Jive (Java Interactive Verification Environment). The tool enables its user to formally prove that a given Java program behaves as expected relative to interface specifications.

Currently there is an ongoing project which aims at changing the specification language of Jive from first-order logic to a specification language that is specifically tailored to Java programs. This change will ease the task of writing interface specifications for programmers who do not have a solid background in mathematics or logic.

The specification language, called Java Modelling Language (JML), uses enriched Java expressions to specify properties of programs via pre- and post-conditions for methods and invariants for classes. Due to many similarities with Java the overhead of learning JML is negligible for programmers compared to other specification languages based on first-order logic. Specifications have to be precise and abstract. Therefore, so-called specification-only model classes are used to provide a mathematical model for data types such as sequences, sets, etc.

Instead of such model classes, the Jive tool works with abstract data types specified in predicate logic. To be able to use JML specifications in Jive, there has to be a corresponding abstract data type for each model class in the JML Library. The goal of this paper is to develop and show how these model classes could be constructed, taking as example the two model classes JMLMath and JMLObjectSequence.

1.2 Aims

The goals of this work could be summarized as follows:

- Find an adequate representation of the JML model classes in a predicate logic
- Reflect as much aspects as possible of the JML language in the Isabelle theories
- Reusage of the already predefined theories in Isabelle in the abstract data type definitions
- Try to find a schematical way to construct the abstract data types in order to be able to generate the Isabelle theories automatically in a further step
- Discuss the problems that could arise when trying to transform constructs of object-oriented programming languages to modular specifications

1.3 JML - Java Modeling Language

This section gives the reader a tutorial introduction to JML. For more information take a look at [4] and [1].

1.3.1 The Specification Language

Design by Contract

Design by Contract is a method for developing software. The principal idea behind is that a class and its clients have a "contract" with each other. The client must guarantee certain conditions before calling a method defined by the class, and in return the class guarantees certain properties that will hold after the call. The use of such pre- and postconditions to specify software dates back to Hoare's 1969 paper on formal verification [3].

JML Specification Expressions

The main restriction in JML is that expressions used in JML's assertions cannot have side effects. Thus Java's `=, +=, ++, --`, etc. are not allowed. In addition, only pure methods are allowed in assertions. A method is *pure* if it has no side effects on the program's state. In JML, one must declare a method to be pure by using the `pure` modifier in the method's declaration. Here are further JML extensions to Java expressions:

Syntax	Meaning
`\result`	result of method call
$a \Longrightarrow b$	a implies b
$a \Longleftarrow b$	a follows from b (i.e. , b implies a)
$a \Longleftrightarrow b$	a if and only if b
a <=! => b	not (a if and only if b)
\old(E)	value of E in pre-state

Table 1: JML extensions

Information Hiding in Specifications

JML supports the notion of information hiding by using Java's privacy levels. JML enforces information hiding by ensuring that public specifications can only mention publicly-visible names. That is, JML does not allow private fields to be used in public specifications. Even though Java could consider some fields or methods to be private, with the annotation `spec_public` we can achieve that these names are treated as publicly-visible in JML specifications.

Non-Null Annotations

The `non_null` annotation is a shorthand way of saying that, in every publicly-visible states, the annotated variable is not null.

Invariants

An *invariant* is a property that should hold in all client-visible states. It must be true when control is not inside the object's methods. That is, an invariant must hold at the end of each constructor's execution, and at the beginning and end of all methods.[1] Furthermore, invariants have to hold before and after method calls in a method body. In JML one can also specify invariants with more restrictive visibility; such invariants, which are not visible to clients, are sometimes called *representation invariants*. Representation invariants can be used to define acceptable internal states of an object; for example, that a linked list is circular, or other similar design decisions.

Quantifiers

JML supports various kinds of quantifiers in assertions: a *universal quantifier* (`\forall`), an *existential quantifier* (`\exists`), *generalized quantifiers* (`\sum, \product, \min, \max`), and a *numeric quantifier* (`\num_of`). For example, the following predicate uses a universal quantifier to assert that all students found in the set `juniors` have advisors.

```
(\forall Student s; juniors.contains(s); s.getAdvisor() != null)
```

Inheritance of Specifications

In JML, a subclass inherits specifications such as preconditions, postconditions and invariants from its superclasses and interfaces that it implements. An interface also inherits specifications of the interfaces that it extends. The semantics of specification inheritance reflects that of code inheritance in Java; e.g., a program variable appearing in assertions is statically resolved, and an instance method call is dynamically dispatched. An important feature of JML's specification inheritance is that its semantics supports a behavioral notion of subtyping. The essence of behavioral subtyping is summarized by Liskov and Wing's *substitution property*, which states that a subtype object can be used in place of a supertype's object [5].

[1] However, in JML, constructors and methods declared with the modifier *helper* are exempted from having to satisfy invariants. Such helper methods must be private, and can be used to avoid duplication of code, even code that does not start or end in a state that satisfies a type's invariant.

Model Fields

As a simple example, consider changing a spec_public field's name from

```
private /*@ spec_public non_null @*/ String name;
```

to

```
private /*@ non_null @*/ String fullName;
```

Ideally, you do not want to make a change to your public specifications, as such change may effect the client code because the client relies on the public specification. For specifications, you want to keep the old name public, but for implementation, you do not want to have two fields taking up space at runtime. In JML, you can solve this dilemma by using a model variable. A *model variable* is a specification-only variable. For example you can make the old field a model field, introduce a new field, and define a relationship between them, as shown below.

```
//@ public model non_null String name;
private /*@ non_null @*/ String fullName;
//@ private represents name <- fullName;
```

The `represents` clause, which is also private in this case, defines an *abstraction function* that maps a concrete representation value to an abstract value of the model field.

Ghost Fields

A `ghost` field is also only present for purposes of specification. However, unlike a model field, it does not have a value determined by a `represents` clause, instead its values is directly determined by a `set`-statement.

A Little Example

To conclude, a little example on how to use JML. This example is partially taken from [4].

Figure 1: A class Person using JML annotations

```
public class Person {
   private /*@ spec_public non_null @*/ String name;
   private /*@ spec_public @*/ int weight;

   /*@ public invariant !name.equals("") && weight >= 0; @*/

   /*@ requires n != null && !n.equals("");
     @ ensures n.equals(name)
     @ && weight == 0;
     @*/
   public Person(String n) {
      name =n; weight = 0;
   }

   /*@ ensures \result != null
     @ && (* result is a displayable form of this person *);
     @*/
   public String toString() {
      return "Person(\"" + name + "\"," + weight + ")";
   }


   //@ ensures \result == weight;
   public int getWeight() {
      return weight;
   }

   /*@ ensures kgs > 0
     @ && weight == \old(weight) + kgs;
     @ signals (Exception e) kgs < 0
     @ && e instanceof IllegalArgumentException;
     @*/
   public void addKgs(int kgs) {
      if (kgs >= 0) {
         weight += kgs;
      } else {
         throw new IllegalArgumentException();
      }
   }
```

1.3.2 JML Model Classes

JML comes with a suite of pure types, implemented as Java classes, that can be used as conceptual models in detailed design. These classes are called model classes. A model class needs not be implemented, but is used only for specification purposes. Since it is pure, none of its methods can permit side-effects.

1.4 Isabelle/HOL

Giving an introduction to Isabelle/HOL in one section is quite impossible. [6] is a good introductory tutorial to the Isabelle/HOL system. Nevertheless, this section has some information and a very short introduction.

The theorem prover Isabelle/HOL can be used as a specification and verification system. Isabelle, which is implemented in ML, is a generic system for implementing logical formalisms, and Isabelle/HOL is the specialization of Isabelle for HOL, which abbreviates Higher-Order Logic. HOL can represent most mathematical concepts, and functional programming is just one particularly simple and ubiquitous instance.

1.4.1 Theories

Working with Isabelle means creating theories. Roughly speaking, a theory is a named collection of types, functions, and theorems, much like a module in a programming language or a specification in a specification language. In fact, theories in HOL can be either. The general format of a theory `T` is

$$\textit{theory}\, T = B_1 + \ldots + B_n : \ \textit{declarations},\ \textit{definitions and proofs end}$$

where $B_1, \ldots, B_n$ are the names of existing theories that T is based on and *declarations, definitions and proofs* represents the newly introduced concepts (types, functions, etc.) and proofs about them. The B_i are the direct parent theories of T. Everything defined in the parent theories (and their parents, recursively) is automatically visible. To avoid name clashes, identifiers can be qualified by theory names as in $T.f$ and $B.f$. Each theory T must reside in a theory file named T.thy. A complete grammar of the basic constructs is found in the Isabelle/Isar Reference Manual [7].
HOL's theory collection is available online at

`http://isabelle.in.tum.de/library/HOL`

and is recommended browsing.

There is also a growing library [2] of useful theories that are not part of `Main` (the union of all the basic predefined theories like arithmetics, lists, sets, etc.), but can be included among the parents of a theory and will then be loaded automatically.

1.4.2 A Little Theory Example

This section is intented to give the reader a look into Isabelle theories and proofs. A theory `ToyList` is defined which represents a small fragment of the already predefined HOL theory `List`. After defining our lists, we will try to prove that reversing a list L twice produces the original list L. Lemmas needed to prove the stated theorem are defined, too. For those who aren't familiar with ML like programming languages or Isabelle: further explanations can be found in [6].

Figure 2: A Theory of Lists

```
theory ToyList = PreList:

datatype 'a list = Nil                          ("[]")
                 | Cons 'a "'a list"           (infixr "#" 65)

consts app :: "'a list => 'a list => 'a list" (infixr "@" 65)
       rev :: "'a list => 'a list"

primrec
"[] @ ys       = ys"
"(x # xs) @ ys = x # (xs @ ys)"

primrec
"rev []       = []"
"rev (x # xs) = (rev xs) # (x # [])"
```

Figure 3: Proofs about lists

```
lemma app_Nil2 [simp]: "xs @ [] = xs"
apply(induct_tac xs)
apply(auto)
done

lemma app_assoc [simp]: "(xs @ ys) @ zs = xs @ (ys @ zs)"
apply(induct_tac xs)
apply(auto)
done

lemma rev_app [simp]: "rev(xs @ ys) = (rev ys) @ (rev xs)"
apply(induct_tac xs)
apply(auto)
done

theorem rev_rev [simp]: "rev(rev xs) = xs"
apply(induct_tac xs)
apply(auto)
done
```

1.5 General Restrictions of the Modelling

For each of the applied modelling approaches, the following restrictions and guidelines have to be fullfilled. These guidelines are not just made in the prospective to make our life easier, furthermore they should help to understand the very important aspects and the essentials of the transformation.

Exceptional Behaviour of the methods is not modeled in the abstract datatype definitions. The reason for underspecification is that our semantics yields undefined for abruptly terminating assertions.

Privacy Levels of JML are not taken into consideration. Suppose we have an invariant inv_i that is declared to be private. Trivially inv_i has to hold, regardless of its visibility state. In Isabelle we do not have the possibility to map this kind of information hiding.

Subtyping is not modeled in the Isabelle theories. Suppose we have a class T which has a method called *equals* (maybe inherited from the base type *Object*) that takes as argument a variable o of type *Object*. A mapping to Isabelle would result in a function declaration $equals :: T \rightarrow Object \rightarrow bool$. As Isabelle does not directly support the subsitution principle of object-oriented programming languages, we have no possibility to easily map such methods.

Type Hierarchy of JML has not to be directly reflected in Isabelle.

Further restrictions regarding the JML language itself are

- `represents` clauses are not modeled
- `assignable` clauses are not modeled
- `ensures_redundantly` clauses are not modeled if they contain informal statements, e.g. "(* the list contains exactly the elements x,y *)"
- `for_example` clauses are not modeled
- constants used only for any status information are not modeled,
  e.g. `public static final String EMPTY_MESS = "This sequence is empty"`

2 Model Class JMLMath

2.1 General Remarks

As stated earlier, *JMLMath* consists only of static methods. Although *JMLMath* inherits from *Object,* its constructor is private so nobody can instantiate it. This has the following consequences for our mapping:

- model and ghost fields inherited from *Object* do not have to be mapped
- no need to model methods inherited from *Object*

As the difficulty level of mapping *JMLMath* is not that high, we can also concentrate on another interesting topic. Let us try not only to find a mapping to Isabelle of our methods, pre- and postconditions, but to show in addition, that the JML specification is correct. As we will see, interesting results can be extracted.

Due to the rapid development of JML, some specifications may have changed. The version of JML at the time of writing is *5.0*.

2.2 Predefined Isabelle Theories

The use of reals suggests that instead of the standard parent theory `Main`, we should make use of `Complex_Main`, which contains a huge quantity of additional functions, theorems and lemmas concerning reals. `Complex_Main` resides in the `HOL-Complex` logic.

2.3 Transformation

Having defined the scope of our work on *JMLMath*, we can already write the first line of our theory:

```
theory JMLMath = Complex_Main:
```

2.3.1 Inheritance

As already observed in section 2.1, *JMLMath* inherits from class *Object*, but due to the impossibility of instantiating *JMLMath*, no mapping has to be applied.

2.3.2 Fields

Model and ghost fields are all inherited from *Object*. Thus the same arguments as in 2.1 and in the previous section hold.

2.3.3 Invariants

This class contains no invariants.

2.3.4 Methods

Thanks to the huge quantity of predefined theories that the Isabelle system offers, a lot of methods in *JMLMath* do not have to be reimplemented. The following list shows which one they are.

- `min`, overloaded constant in HOL
- `max`, overloaded constant in HOL
- `abs`, overloaded constant in HOL
- `pow`

- `sqrt`

The functions marked to be overloaded in HOL are applicable to both integers and reals. In the section about pre- and postconditions we will only specify the postcondition of `sqrt` as for the others listed above the postcondition exactly corresponds to the function definition.

Now, let's have a look at the other methods. In this section we will only try to give a definition and if needed an implementation of the methods in *JMLMath*. The mapping is rather straight forward, but each method will be discussed in detail.

ceil

declaration: `public static \real ceil(\real a)`

returns: the smallest (closest to negative infinity) `\real` value that is not less than the argument and is equal to a mathematical integer

Fortunately, Isabelle has a function called `ceiling` that returns an integer instead of a real. Thus, the resulting definition is

```
ceil :: "real => real"
"ceil a == real (ceiling a)"
```

floor

declaration: `public static \real floor(\real a)`

returns: the largest (closest to positive infinity) `\real` value that is not greater than the argument and is equal to a mathematical integer

As for `ceil`, Isabelle also offers a predefined function `floor` that returns an integer. To avoid name clashes, we just rename our function to `rfloor`

```
rfloor :: "real => real"
"rfloor a == real (floor a)"
```

Another possibility to solve this problem would be to specify our function with the fully qualified name. The predefined function resides in the Isabelle theory `IntFloor`.

```
floor :: "real => real"
"floor a == real (IntFloor.floor a)"
```

Clearly this solution is more elegant and in the future we will stick to this one, if this problem should arise one more time.

nearestInteger

declaration: `public static \bigint nearestInteger(\real a)`

returns: the nearest `\bigint` value to the argument. If a is a value of the form $x.5$, then a is rounded to the even.

For the function `nearestInteger` we choose a different approach as for the other functions. A concrete implementation would be somehow too ugly and would make it quite complicated when trying to define lemmas about the postcondition. Instead, we choose an axiomatic definition. Isabelle gives us the possibility to work with *definite descriptions*. This concept will be formalized in section 4.2.1 on page 27, as it turns out to be very useful regarding the automated transformation of model classes. At this point of the paper, the definition is given whithout further explanations. Just note the use of the THE-operator and that the definition corresponds to the postcondition of `nearestInteger`.

```
nearestInteger :: "real => bigint"
"nearestInteger a == THE n::bigint.
                           abs (real n - a) <= ((1/2)::real) &
                           (abs(real n -a) = ((1/2)::real) --> n mod 2 = 0)"
```

rint

declaration: `public static \real rint(\real a)`

returns: the nearest `\real` value to the argument. If a is a value of the form $x.5$, then a is rounded to the even.

After having defined `nearestInteger`, the definition of `rint` is trivial.

```
rint :: "real => real"
"rint a == real (nearestInteger a)"
```

round

declaration: `public static \bigint round(\real a)`

returns: the value of the argument rounded to the nearest `\bigint` value, i.e.

$$JMLMath.floor(a + 0.5d)$$

Reading the specification, the definition can be written down straight forward.

```
round          :: "real => bigint"
"round a == IntFloor.floor (a + (1/2)::real)"
```

The use of `Intfloor.floor` instead of `JMLMath.floor` is justified by the fact that round returns a bigint value and not a real value.

2.3.5 Pre- and Post-Conditions

As mentioned before, no lemmas regarding the methods `min, max, abs` and `pow` have to be defined, as the JML specification exactly corresponds to the function definition in Isabelle.

For the other methods in $JMLMath$, the following guidelines will be respected:

- Each lemma will be named in the following manner: `post_functionName`
- Each lemma will be added to Isabelle as a simplification rule by the keyword `[simp]`

sqrt

First, let's have at look a the specification. We see that no precondition is defined (lack of the keyword `requires`)

```
@ public normal_behavior
@ ensures \result > 0 && \result * \result == a;
```

As this is our first lemma we want to prove, the postcondition will be decomposed into the two conjuncts. The first lemma we can define is

```
lemma post_sqrt_gt_zero [simp]: "sqrt a > 0"
```

Let Isabelle try to prove this lemma. After trying with some commands in Isabelle, we conclude that there is no chance that Isabelle succeeds to prove this lemma, neither with simplification (`by simp`) nor by the `auto` tactic. There is one simple reason: no precondition is given. Suppose that a is negative. We would get a complex number, which are not represented in $JMLMath$, or an exception would be risen. We have to exclude negative arguments. Thus, we can conclude that the JML specification is incomplete. Adding the precondition to the lemma has the nice effect that Isabelle can easily solve this lemma by applying a predefined rule.

```
lemma post_sqrt_gt_zero [simp]: "0 <= a ==> 0 <= sqrt a"
apply(rule Transcendental.real_sqrt_ge_zero)
apply(auto)
done
```

Now, we shall concentrate on the second conjunct. With the conclusions gained above, we can state our lemma and let Isabelle prove it by simplification.

```
lemma post_sqrt_mult_equals [simp]: "0 <= a ==> sqrt a^2 = a"
by simp
```

As we can see, we added the precondition from above to the lemma. So, we have finished the proof regarding the postcondition of `sqrt`.

ceil

```
@ public normal_behavior
@  ensures (* \result == TBC *);
```

We can only suppose what TBC means. Maybe it should represent something like a random number. As we cannot really do a lot with this postcondition, we just define our own postcondition

```
@ public normal_behavior
@  ensures \result >= a;
```

It is evident that this condition is weak. We yield it to the reader to look for a more precise postcondition. The resulting lemma is

```
lemma post_ceil [simp]: "a \<le> ceil a"
by simp
```

and is easily proved by simplification.

floor

```
@ public normal_behavior
@  ensures (* \result == TBC *);
```

Again, we define a postcondition by ourselves

```
@ public normal_behavior
@  ensures \result <= a;
```

and state the lemma, proved by Isabelle by simplification again.

```
lemma post_floor [simp]: "floor a \<le> a"
by simp
```

nearestInteger

```
/*@
  @ public normal_behavior
  @  ensures abs(\result - a) <= 0.5 &&
  @          abs(\result - a) == 0.5 ==> \result % 2 == 0;
  @*/
```

As discussed in section 2.3.4 on page 12, the definition of `nearestInteger` already corresponds to its postcondition. Consequently, no further lemmas have to be defined. One thing we could do is to add the definition of `nearestInteger` to the simplification rules set of Isabelle. This is done by the command

```
declare nearestInteger_def [simp]
```

rint

```
/*@
  @ public normal_behavior
  @  ensures \result == nearestInteger(a);
  @*/
```

This is a similar case to `nearestInteger`. We can either declare the definition as simplification or state a lemma and prove it. We will do both

```
declare rint_def [simp]

lemma post_rint [simp]: "real (nearestInteger a) = rint a"
by simp
```

round

Here again, the definition is equal to the postcondition.

```
declare round_def [simp]
```

2.4 Summary

Defining the function and lemmas for $JMLMath$ was not a big effort. One reason is surely that $JMLMath$ contains only static definitions and we did not need to worry about the dynamic aspects. We were able to reflect all definitions in $JMLMath$ in the Isabelle theory. The only thing that someone has to pay attention to, is that Isabelle gets easily stuck when dealing with real numbers. Proof techniques like *induction* and *case distinction* cannot be easily applied. But with the continuing development of the Isabelle system (currently Isabelle2004) more and more rules and definitions concerning reals are added which should simplify the work with them.

Another important point of view is, that with Isabelle we did not just map the $JMLMath$ model class to an abstract data type definition. In some cases we could also prove that the JML specification was not correct.

Thanks to the closeness to mathematics of this model class, our aim to reuse as much predefined theories as possible could be achieved quite successfully because of the richness of predefined theories in the Isabelle system.

Generating $JMLMath$ automatically would have been quite difficult, if we wanted to transform it in a way, such that we had the available implementation. With another approach, e.g. *axiomatic specification*, the automated transformation would have been feasible.

3 Model Class JMLObjectSequence

3.1 General Remarks

In the following sections we will try to find an appropriate mapping for the model class *JMLObject-Sequence*. Due to the largeness of this class we renounce to show the mapping of each method. The mapping of a set of chosen methods will be shown in order to cover as much diversities as possible.

Modelling *JMLObjectSequence* involves a lot of new concepts compared to *JMLMath*. One important aspect is the instantiation of *JMLObjectSequence* objects, meaning that we have to take account of dynamic method binding and object creation. Each method is bound to an object and not to a class. As Isabelle does not directly support the notion of classes and objects, we have to look for some workarounds. The first concerns method invocation. Suppose we have an object o of Type O and we want to call a method m on o. In object-oriented programming we would have a method call like *o.m(arg1,arg2)* whereas in Isabelle we have to pass the actual object and the method call results in *m(o,arg1,arg2)*. In section 4 on page 27 we will define a small set of rules concerning the transformation of methods.

The second problem is that we have to find a representation for class fields. Therefore two modelling approaches will be discussed in this section. The first model is rather simple and does not take respect of mapping class fields to a corresponding representation in Isabelle. This surely is a rather strong restriction but can be helpful in the future. The second approach will be more sophisticated and will map objects into tuples of attributes. Induction and inductively defined sets will play an important role.

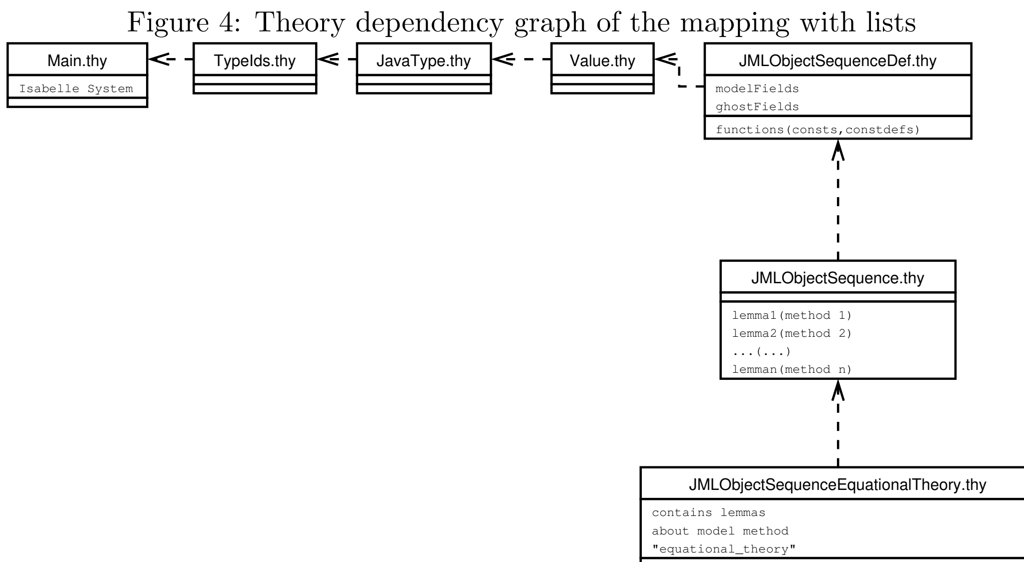
3.2 Predefined Isabelle Theories

As mentioned before, our first approach makes use of lists. The Isabelle standard parent theory `Main` already contains all important theories and no special treatment has to be applied for our theories. Just note that the majority of the used functions are situated in `List.thy`.

3.3 First Approach: Modelling the Sequence as Primitive List

3.3.1 Overview

Figure 4: Theory dependency graph of the mapping with lists



On this overview we can see the theory dependency of our first approach. `Main, TypeIds, JavaType` and `Value` are of no interest to us. `JMLObjectSequenceDef.thy` contains the definitions (model fields, ghost fields and methods) of our mapping. In `JMLObjectSequence.thy` reside our pre- and post-conditions of the methods defined in the parent theory. `JMLObjectSequenceEquationalTheory.thy` contains lemmas

concerning the model method *equational_theory(JMLObjectSequence s, JMLObjectSequence s2, Object e1, Object e2, int n).* This method is an equational specification of the properties of sequences meaning that all *JMLObjectSequence* objects have to satisfy this method, otherwise the object is not a *JMLObjectSequence.*

3.3.2 Restrictions

Due to the representation as lists, we have some restrictions on our model:

Definitions

- Fields that should hold a value are not mapped
- Static fields, that hold a value of a method, e.g. the field *EMPTY*, are mapped
- Ghost fields are not mapped

Pre- and Post-Conditions

- Conditions involving model and ghost fields can not be mapped

Equational Theory

- Properties involving ghost fields can not be mapped

3.3.3 Definitions

Representation

Let us first define the type of our sequences

```
types JMLObjectSequence_rep = "value list"
```

This kind of representation gives us the possibility to work with the huge amount of predefined functions in the theory `List.thy`, but we have some drawbacks as mentioned in the sections above.

Fields

Apart from *EMPTY* no field has to be mapped. Having a look at the java specification and implementation of *EMPTY* we see that it represents just an empty sequence.

```
public static final /*@ non_null @*/ EMPTY = new JMLObjectSequence();
```

This leads us to following Isabelle implementation:

```
constdefs
EMPTY :: "JMLObjectSequence_rep"
"EMPTY == []"
```

Constructors

JMLObjectSequence offers three constructors. The first one is the standard constructor

```
public JMLObjectSequence() {
        //@ set owner = null;
        theSeq = null;
        length = 0;
        //@ set elementType = \type(Object);
        //@ set containsNull = false;     }
```

Because we do not have to care about ghost fields and model fields in this mapping, we only have to consider line *3*, which leads to the following Isabelle mapping

```
JMLObjectSequence_stdcons :: "JMLObjectSequence_rep"
"JMLObjectSequence_stdcons == EMPTY"
```

The second constructor instantiates a sequence with just one element. Translated into our mapping, a function returning a list has to be provided

```
singleton :: "value => JMLObjectSequence_rep"
"singleton xs == xs#[]"

JMLObjectSequence_cons2 :: "value => JMLObjectSequence_rep"
"JMLObjectSequence_cons2 xs == singleton xs"
```

Due to the restrictions defined above - no mapping of fields holding a value - there is no possibility to map the third constructor

```
protected JMLObjectSequence (JMLListObjectNode ls, int len){...}
```

Methods

Basically we have to distinct two cases:

1. Method must be implemented
2. Method must be implemented, but we can make use of library functions or reuse of self-defined functions

1. Method must be implemented

We have no other choice then to take a closer look at the specification and implementation of the concerned methods and do the mapping by hand. Due to the fact, that this work involves just some founded knowledge in functional programming, the mapping of a subset of *JMLObjectSequence*'s methods is demonstrated here

JMLObjectSequence - the mapping of some methods

```
consts
   count              :: "JMLObjectSequence_rep => value => int"
   containsAll        :: "JMLObjectSequence_rep => JMLObjectSequence_rep => bool"
   isPrefix           :: "JMLObjectSequence_rep => JMLObjectSequence_rep => bool"
   isInsertionInto    :: "JMLObjectSequence_rep => JMLObjectSequence_rep => value => bool"


primrec
"count [] x = 0"
"count (h#t) x = (if h=x then
                    1 + (count t x)
```

```
                  else
                    count t x
                  )"

primrec
"containsAll [] c = True"
"containsAll (h#t) c = (if (h mem c) then
                          (containsAll t c)
                        else
                           False
                        )"

primrec
"isPrefix xs [] = (case xs of
                     [] => True
                   | z#zs => False)"
"isPrefix xs (y#ys) = (case xs of
                         [] => True
                       | z#zs => (if (z=y) then
                                    (isPrefix zs ys)
                                  else
                                    False
                                  )
                       )"

primrec
"isInsertionInto xs [] ys = ((hd(xs)=ys) & (length xs)=1)"
"isInsertionInto xs (h#t) ys = (case xs of
                                  [] => False
                                | h1#t1 => (if h1=ys then
                                             ((isPrefix t1 t) & (containsAll t1 t))
                                            else
                                              (if h1=h then
                                                (isInsertionInto t1 t ys)
                                               else
                                                 False
                                               )
                                            )
                                )"

constdefs
insertFront :: "JMLObjectSequence_rep => value => JMLObjectSequence_rep"
"insertFront l x == (x#[]) @ l"

insertBack :: "JMLObjectSequence_rep => value => JMLObjectSequence_rep"
"insertBack l x == l @ (x#[])"
```

2. Method must be implemented, but we can make use of library functions or reuse of self-defined functions

In this case the work mainly consists of going through the Isabelle theory `List.thy` and determining which function can be used for the according mapping. Let's have a look at some of this methods.

itemAt

declaration: `public Object itemAt(int i) throws JMLObjectSequenceException`

returns: the element at the given zero-based index.

mapping: Fortunately the `List` theory holds a definition of function `nth` which takes a natural number n and returns the element at the given index n. Because the Java method expects an integer as argument, we have to convert the integer argument of `itemAt` to a natural number and pass it to `nth`. Therefore Isabelle provides a function `nat` which takes an integer and converts it to natural number. This leads us to the following implementation

```
itemAt :: "JMLObjectSequence_rep => int => value"
```

```
"itemAt l i == nth l (nat i)"
```

Seems to be a reasonable solution but we can get in trouble when looking at the following lemma

```
lemma: "itemAt l -2 = itemAt l 0"
```

which should clearly be unsolvable, but Isabelle can solve it! The problem is the function `nat`. According to the definition of `nat`, the expression `nat i` evaluates to 0, $\forall i. i < 0$. This means that we have to provide a function allowing only inputs greater or equal to zero. Isabelle therefore offers the concept of *underdefined functions*. If the input is less than zero then the function is not executed. Obviously, there are very nice effects when thinking about the fact, that we do not want to model exceptional behaviour.

```
constdefs myNat:: "int => nat"
"0 \<le> i ==> myNat i \<equiv> nat i"
```

and the new definition of `itemAt` is

```
itemAt :: "JMLObjectSequence_rep => int => value"
"itemAt l i == nth l (myNat i)"
```

concat

declaration: `public JMLObjectSequence concat(/*@ non_null @*/ JMLObjectSequence s2)`

returns: Return a sequence that is the concatenation of this with the given sequence

mapping: Due to the facilities when working with lists in Isabelle, the mapping results to be quite easy.

```
concat :: "JMLObjectSequence_rep => JMLObjectSequence_rep => JMLObjectSequence_rep"
"concat t s == (t @ s)"
```

removeItemAt

declaration: `public /*@ non_null @*/ JMLObjectSequence removeItemAt(int index)`

returns: Return a sequence like this, but without the element at the given zero-based index

mapping: `removeItemAt` looks more complicated as the other but in the end, making use of predefined theories eases the work

```
removeItemAt :: "JMLObjectSequence_rep => int => JMLObjectSequence_rep"
"removeItemAt x i == (take (myNat i) x) @ (drop ((myNat i) + 1) x)"
```

3.3.4 Pre- and Post-Conditions

The first discussion should be kept in regards to the type of mapped functions. Properties of recursively defined functions are best established by induction. In the case. where we have to prove lemmas involving recursive functions, there is nothing obvious except induction on the list. Anyway, some cases will be encountered where the lemmas are easier proved by simplification, although recursively defined functions will be involved.

As mentioned earlier we have no possibility to map or state lemmas, where ghost fields are part of the pre- or postcondition. Some examples of lemmas will be shown in the next paragraph. We will first

concentrate on those, which are solvable quite straight forward and then continue with a few, which are not that simple or seemingly impossible to prove.

It's important to be aware of the fact, that all constant definitions have been added to the set of simplification rules in `JMLObjectSequenceDef.thy`. This is advantageous when applying the simplification tactic to a lemma.[2]

post_singleton[3]

JML-Specification:

```
/*@ public normal_behavior
  @    assignable \nothing;
  @    ensures \result != null && \result.equals(new JMLObjectSequence(e));
  @*/
```

Isabelle-Lemma:

```
lemma post_singleton: "singleton e \<noteq> [] & singleton e = JMLObjectSequence_cons2 e"
by simp
```

Explanation/Comments: `singleton` has no precondition and the lemma is solved directly by simplification. `singleton e` in the first conjunct is simplified to `e#[]-->[e]` so that $[e] \neq []$ holds $True$. `JMLObjectSequence_cons2 e` is equal to `singleton e` according to its definition, so that the second conjunt holds $True$ and the whole expression is $True$.

post_convertFrom

JML-Specification:

```
/*@ public normal_behavior
  @    requires a != null;
  @    assignable \nothing;
  @    ensures \result != null && \result.size() == a.length
  @         && (\forall int i; 0 <= i && i < a.length;
  @                         (\result.itemAt(i) == null
  @                            ? a[i] == null
  @                            : \result.itemAt(i) == a[i]));
  @*/
```

Isabelle-Lemma:

```
lemma post_convertFrom: " a\<noteq> [] ==> size (convertFrom a) = length a
                                            & (
                                               \<forall> (i::int). 0 <= i & i < int (length a) -->
                                                   (
                                                     if (itemAt (convertFrom a) i) = Null then
                                                       (itemAt a i) = Null
                                                     else
                                                      (itemAt (convertFrom a) i) = (itemAt a i)
                                                    )
                                                )"
by simp
```

Explanation/Comments: This is a very interesting example of a lemma that Isabelle solves with (only) simplification, although the lemma looks very complicated at first sight.

[2] *apply(simp)* can be used instead of being obliged to manually add the simplication rule by *apply(simp add: someConstDef, someOtherConstDef)*

[3] recall the naming conventions defined in 2.3.5 on page 13

post_count

JML-Specification:

```
/*@
  @   public normal_behavior
  @     requires item != null;
  @     ensures \result
  @         == (\num_of int i; 0 <= i && i < length();
  @                             itemAt(i) != null
  @                             && itemAt(i) == item);
  @ also
  @   public normal_behavior
  @     requires item == null;
  @     ensures \result == (\num_of int i; 0 <= i && i < length();
  @                                        itemAt(i) == null);
  @*/
  //@ implies_that
  //@    ensures \result >= 0;
```

Isabelle-Lemma:

```
lemma post_impl_count :"0 \<le> (count x i)"
apply(induct_tac x)
by auto
```

Explanation/Comments: Because the two *normal_behavior* cases are too difficult to map, only the implication of the specification is used. Induction on x can be applied and the tactic `auto` then does the rest.

post_trailer

JML-Specification:

```
/*@  public normal_behavior
  @    requires length() >= 1;
  @    ensures \result.equals(removePrefix(1));
  @    ensures_redundantly \result.length() == length() - 1
  @           && (* \result is like this, but without the first item *);
  @ also
  @  public exceptional_behavior (...)
  @*/
  //@implies_that
  //@    ensures !\result.containsNull <== !containsNull;
```

Isabelle-Lemma:

```
lemma post_trailer: " 1 \<le> length this ==> equals (trailer this) (removePrefix this 1)"
```

Isabelle-Lemma (redundant):

```
lemma post_trailer_redund: "1 \<le> length this ==> length (trailer this) = ((length this) - 1)"
apply(simp)
apply(case_tac this)
apply(auto)
done
```

Explanation/Comments: We can not model the implication because `containsNull` is a ghost field. So we focus on the postcondition when no exceptional behavior is expected. Although the lemma looks easy, it can not be solved. Maybe some further lemmas had to be defined. This is in contrast with the redundant specification, which was solved straight forwardly making use of a new tactic `case_tac`, which performs case distinction on `this`. To look at further lemmas that were not solvable, e.g. `post_concat`, consult C.2 on page 35.

3.3.5 Equational Theory

The precise definition of *equational theory* of a class of structures corresponds to the set of universal atomic formulas that hold in all members of the class. Suppose we have n universal atomic formulas F_i, $i = 1 \ldots n$ and a method *equational_theory(someArgs)* of a class C, and $equational_theory \equiv F_1 \wedge \ldots \wedge F_n$. If $c \in C$ then $c.equational_theory(\ldots)$ holds for all valid inputs of $equational_theory$. For our transformation of this model method, we can proceed as follows:

1. Take the definition of the model method
2. $\forall F_i \in equational_theory_definition$ generate an according lemma $post_F_i$ and prove it

As the lemmas are short and compact in most cases, no lemma of the equational theory will be proved here. For those interested, view C.2 on page 42.

3.3.6 Summary

The approach with primitive lists makes life easier because the difficulty level is lowered. But some drawbacks can not pass unobserved. The mapping is rather easy to understand but incomplete. The fact, that model fields are not modeled, has in some cases the consequence, that even some method specifications can not be modelled because these specifications rely on them.

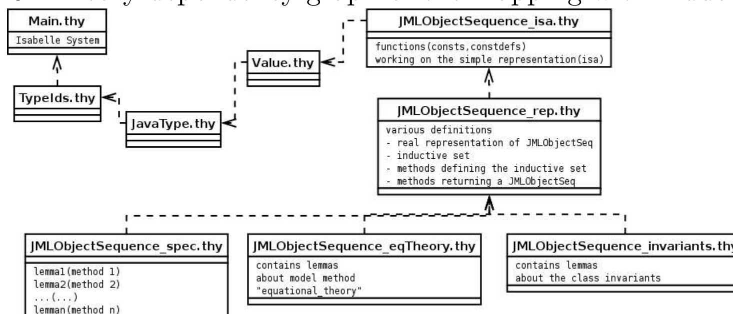
Due to the complexity and the length of some specifications, the proofs did not sometimes go off without a hitch and working with the theorem prover became very tedious.

The automated transformation of *JMLObjectSequence* using this approach seems to be impossible, unless we would have an intelligent Isabelle theory-parser which is able to decide, if a library function could be integrated into an abstract datatype definition of such a model class.

3.4 Second Approach: Inductively Defined Sets

3.4.1 Overview

Figure 5: Theory dependency graph of the mapping with inductive sets



As in 3.3.1 on page 17, `Main.thy`, `TypeIds.thy`, `JavaType.thy` and `Value.thy` are part of the Isabelle system or of the JIVE framework and do not need further explanations. `JMLObjectSequence.thy` represents the whole model class in Isabelle and has the following form

```
theory JMLObjectSequence = JMLObjectSequence_spec + JMLObjectSequence_eqTheory + JMLObjectSequence_invariants:

end
```

meaning that it unifies all lemmas and definitions coded in the three other theories and their parent theories.

`JMLObjectSequence_rep.thy` is the key theory of this mapping. The ideas behind are explained in 3.4.3.

JMLObjectSequence_isa.thy is similar to JMLObjectSequenceDef.thy in our first approach and contains the mapping of the methods of *JMLObjectSequence.* The mapped methods are functions working on lists.

JMLObjectSequence_spec.thy, -_eqTheory.thy, -_invariants.thy respectively contain the pre- and postconditions, the model method *equational_theory* and the class invariants of *JMLObjectSequence*.

3.4.2 Restrictions

Due to the usage of a tuple for the attributes [4], we have the possibility to deal with model and ghost fields. The main restriction remains the fact that we can not bind a method to a specific object which in our case is the Isabelle record. Therefore we have to pass the reference of the actual object as already stated in 3.1 on page 17.

3.4.3 Objects, Classes and Inductively Defined Sets

A set is inductively defined when it is generated from some base elements using some set of constructor operations. In the case of objects this base elements are the constructors of the class. All methods returning the same object type *JMLObjectSequence* could also be added to the inductively defined set, but as they often rely on one of the constructors, it suffices to have the constructors as base elements of the set. They completely define the set.

After having defined the main representation of the *JMLObjectSequence* as in 3.4.4, the constructors have to be mapped.

JMLObjectSequence - standard constructor

```
public JMLObjectSequence() {
     //@ set owner = null;
     theSeq = null;
     length = 0;
     //@ set elementType = \type(Object);
     //@ set containsNull = false;
 }
```

Note that we have to take account of the JML @set statements in that case, as they initialize the state of the ghost fields.

```
JMLObjectSequence_stdcons :: JMLObjectSequence_rep
"JMLObjectSequence_stdcons \<equiv> (| getClass = 0,
                                      objectState = 0,
                                      theString = 0,
                                      objectTimesFinalized = 0,
                                      owner = Null,
                                      containsNull = False,
                                      elementType = ClassT Object,
                                      theLength = 0,
                                      theSeq = EMPTY_isa|)"
```

EMPTY_isa returns an empty list. The types of the fields getClass, objectState, theString are declared as dummy types because these fields are not relevant for the mapping.

The mapping of the other constructors can be consulted in C.3 on page 47. Having mapped our constructors, we can define the inductive set in Isabelle. Preconditions can, and shall, be included in this definition, e.g as in line 7.

JMLObjectSequence - the inductive set

```
1  consts ObjSeqSet :: "JMLObjectSequence_rep set"
2
3  inductive ObjSeqSet
4  intros
5  std_constructor[intro!]: "JMLObjectSequence_stdcons \<in> ObjSeqSet"
```

[4]the Isabelle construct for a tuple is *record*

```
6  constructor2   [intro!]: "JMLObjectSequence_cons2 e \<in> ObjSeqSet"
7  constructor3   [intro!]: "((s1 = []) = (length s1 = 0) \<and> (0 \<le> length s1)) ==>
8                                  JMLObjectSequence_cons3 s1 (int (length s1)) \<in> ObjSeqSet"
```

3.4.4 Model and Ghost Fields

For an object of type T with k fields $f_1, \ldots, f_k$ we can easily define such a record

$$record\, T_{rep} = f_1 :: t_{f_1} \;\ldots\; f_k :: t_{f_k}$$

where t_{f_i} is the type of field f_i. This results in following declarations in Isabelle

```
record ModelFields = getClass   :: Class
                     objectState :: JMLDataGroup
                     theString   :: String
record GhostFieldsAndModelFields = ModelFields +
                     objectTimesFinalized :: int
                     owner                :: value
                     containsNull         :: bool
                     elementType          :: javatype

record JMLObjectSequence_rep = GhostFieldsAndModelFields +
                     theLength :: int
                     theSeq    :: JMLObjectSequence_isa
```

where *JMLObjectSequence_isa* corresponds to the type *JMLObjectSequence_rep* of the first approach with lists. The whole record could also have been declared in one single go, whithout extending one redord to another. But for obvious reasons, this version is more meaningful.

3.4.5 Invariants

Thanks to the model of inductive sets, we can easily state lemmas for the invariants. Here we have two invariants of *JMLObjectSequence*. The first one states that the attribute `owner` is always null. Isabelle manage to solve it with the following commands.

```
lemma owner_is_null: "s \<in> ObjSeqSet ==> owner s = Null"
apply(erule ObjSeqSet.induct)
apply(simp_all)
done
```

The next invariant says, that the value of the attribute `length` is always consistent with the actual length of the representation. The function `rep_to_isa` returns the "low-level" representation of our sequence which in this case is the field *theSeq* of the tuple *JMLObjectSequence_rep*.

```
lemma length_is_consistent: "s \<in> ObjSeqSet ==> theLength s = int (length (rep_to_isa s))"
apply(erule ObjSeqSet.induct)
apply(simp_all)
done
```

Almost all invariants can be solved using this approach: If an object o is an object of type *JMLObjectSequence* ($o \in ObjSeqSet$), then the invariant has to hold and induction over the inductive set can be applied.

3.4.6 Summary

Thanks to the approach of modelling the objects with the concepts of induction and inductively defined sets, almost all the important aspects of JML have been mapped into Isabelle/HOL. The well founded theory concerning induction and sets allows one to reason in a mathematical way about objects, which in some cases simplifies things a lot.

Generating automatically a Isabelle theory with this approach would be quite difficult. On one hand problems mentioned already in 3.3.6 on page 24 could arise and on the other hand we have to consider, that a lot of background knowledge is needed.

4 Automated Transformation of JML model classes

4.1 General Remarks

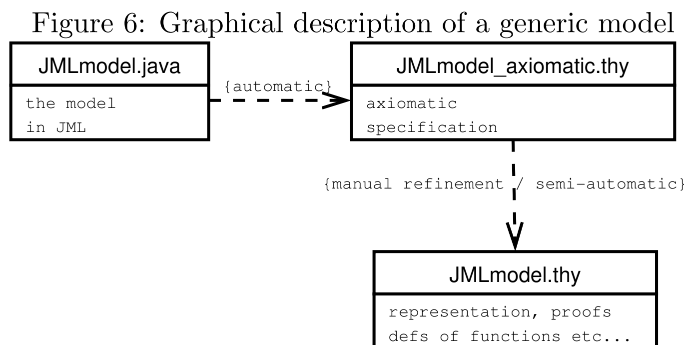
Based on the thoughts made in the previous sections, an approach of generating the Isabelle theories shall be developed. First of all a generic model, which describes how this transformation should be done, has to be defined. But contemporaneously the restrictions of such an approach must be taken into account. The following rigorous condition must be fullfilled in order to guarantee the correctness of this mapping:

- The JML specification of the model classes is correct.

Note that the general restrictions in section 1.5 on page 10 still hold.

4.2 A Generic Model

Figure 6: Graphical description of a generic model



Relying on the correct JML specification, an axiomatic specification of the methods can be generated. How it is done is shown in the next section. Some situations could be encountered, where the axiomatic specification does not suffice. Thus, a refinement needs to be done. This could be achieved in the way it was produced in section 3.4. Nevertheless, a skeleton could be generated which would be of great help for someone translating these model classes. An example of generating these skeletons is demonstrated in section 4.3.

4.2.1 Axiomatic Specification

Assume the existence of a non-static function m with k arguments in a model class *JMLmodel* with precondition PRE_m and postcondition $POST_m$, where $POST_m$ is of the form $\backslash result \Longleftrightarrow POST'_m$.

```
/*@ public normal_behavior
  @    requires PRE_m;
  @    ensures \result <==> POST_m'
  @*/
public retT m(argT_1 arg_1,argT_2 arg_2, ..., argT_k arg_k) { ... }
```

An axiomatic definition of this method can be achieved as follows

```
constdefs
m :: "t_o => argT_1 => argT_2 => ... => argT_k => retT"
"PRE_m ==> m  o a_1 ... a_k == POST_m' "
```

where $argT_i$ is the type of the corresponding argument, t_o the type of the object and $retT$ the one of the return value. If the method m is static, then the first argument can be left out, as the execution of m is not related to an object. Now let's assume that $POST_m$is of a more general form $f(\backslash result, a_1, \ldots, a_k) = b$. Then the mapping to Isabelle can be developed by making use of the `THE` operator of Isabelle

```
constdefs
m :: "t_o => argT_1 => argT_2 => ... => argT_k => retT"
"PRE_m ==> m  o a_1 ... a_k == THE x. f(x, o, a_1, ..., a_k)= b "
```

Since model classes are `pure` by definition and allow no side-effects, the case where *retT* is `void` does not need to be covered.

4.2.2 The Refinement of a Model Class

As mentioned earlier the refinement cannot be done fully automatically. But a skeleton can be provided as it will be shown in the next section. Therefore concepts and ideas gained in this paper will be applied.

4.3 Automated Generation Algorithm

The following algorithm does not aim to be complete as it will not cover all details. It shall serve as a base for new ideas, that could come out of this paper.

Let us assume, we have a model class *JMLmodel* with n methods $m_1, \ldots, m_n$, each of them holding a precondition PRE_{m_i} and a postcondition $POST_{m_i}$, where $1 \leq i \leq n$ and the precondition may be empty. Furthermore, the arguments of m_i are $arg_{1i}, \ldots, arg_{ji}$ of type $argT_{1i}, \ldots, argT_{ji}$ and the return type of m_i is $retT_i$. Moreover, *JMLmodel* has k invariants $INV_1, \ldots, INV_k$ and not compulsory a model method $equational_theory_{model}$.[5] A Isabelle theory collection modelling *JMLmodel* can be generated as follows

1. Generate the axiomatic specification – *JMLmodel_axiomatic.thy*
   (a) Properly define all the types occuring in *JMLmodel.* Sometimes it could be sufficient to state only the existence of a specific type.[6]
   (b) For each m_i in *JMLmodel:*
      i. If m_i is static then the Isabelle type of the method will be
      $$"m_i :: argT_{1i} \rightarrow \ldots \rightarrow argT_{ji} \rightarrow retT_i"$$
      else
      $$"m_i :: o \rightarrow argT_{1i} \rightarrow \ldots \rightarrow argT_{ji} \rightarrow retT_i"$$
      where o represents an object instance of *JMLmodel*, concretely `this`.
      ii. If $POST_{m_i}$ is of the form $\backslash result = f(arg_{1i}, \ldots, arg_{ji})$, the definition of m_i results to
      $$"PRE_{m_i} \Rightarrow m_i\,\{o\}\,arg_{1i} \ldots\, arg_{ji} \equiv f(arg_{1i}, \ldots, arg_{ji})"$$
      iii. Else if $POST_{m_i}$ is of the form $f(\backslash result, arg_{1i}, \ldots, arg_{ji}) = b$, the definition of m_i corresponds to
      $$"PRE_{m_i} \Rightarrow m_i\,\{o\}\,arg_{1i} \ldots\, arg_{ji} \equiv THE\,x.\,f(x, arg_{1i}, \ldots, arg_{ji}) = b"$$
2. Generate the specification concerning the representation on Isabelle level – *JMLmodel_isa.thy*
   (a) Declare appropriately a type $JMLmodel_{isa}$.
   (b) For each method m_i generate a piece of code as described in the next steps
      i. Construct the function declaration as in *step (1.b.i)*

[5] Parent classes and implemented interfaces must also be considered.
[6] Isabelle offers for this purpose the statement *typedecl.*

ii. Generate a stub

$$"PRE_{m_i} \Rightarrow m_i \{o\}\, arg_{1i} \ldots arg_{ji} \equiv "$$

. The precise implementation will be left to the programmer. In some cases, e.g. m_i is recursively defined, this piece of code has to be totally rewritten. Note the included precondition in the function declaration.

iii. If m_i is declared as a constant (`constdefs, defs`, etc.) in Isabelle, then add it to the set of simplification rules.

$$declare\ m_i_def\ [simp]$$

3. Generate the specifications containing the real representation of *JMLmodel – JMLmodel_rep.thy*

   (a) Generate dummy types and/or define them if needed.

   (b) Assume we have l fields in the set $F_{JMLmodel} \equiv MF_{JMLmodel} \cup GF_{JMLmodel}$, where $MF_{JMLmodel}$ is the set of all model fields (parent classes included) and $GF_{JMLmodel}$ the set of all ghost fields in *JMLmodel*. Generate the declaration of the tuple containing the elements of $F_{JMLmodel}$. This will lead to the type declaration

   $$JMLmodel_{rep} = field_1 :: fieldT_1 \ \ \ldots \ \ field_i :: fieldT_l$$

   where $f_i \in F_{JMLmodel}$, $i = 1, \ldots, l$. Note that one of the fields will be of type $JMLmodel_{isa}$.

   (c) Define a function $rep_to_isa :: JMLmodel_{rep} \rightarrow JMLmodel_{isa}$, which returns the Isabelle representation of the tuple defined above. Sometimes it will be more convenient to work with the "low-level" representation of the model class. The function will have the following implementation whith f_i being the field of type $JMLmodel_{isa}$.[7]

      i. `"rep_to_isa theRecord \<equiv> f_i theRecord"`

   (d) Implement the constructors in Isabelle. Therefore, look at the implementation to determine how the values of the fields are set. Note that in Java, the call of the standard constructor automatically determines an execution of the standard constructor of the parent classes. An example, how the constructors are implemented in Isabelle can be seen in C.3 on page 47. It is also important to know, that ghost fields are set by the `\\@set` statement. Shortly, the function definitions of the constructors return a record generated according to the arguments and the implementation.

   (e) Take all the methods m_i, where $retT_i = JMLmodel$ and generate a stub as in *step 2.b*. The implementation of this methods will be based on the constructors because they return a $JMLmodel$ object.

   (f) Add the definitions to the set of simplification rules as in *step 2.b.iii*

   (g) Declare an inductive set $JMLmodelSet :: JMLmodel_{rep}\, set$ and add the constructors to the set of introduction rules of $JMLmodelSet$. This is done as shown below

      i.
```
consts JMlmodelSet :: "JMLmodel_rep set"

inductive JMLmodelSet
intros
constructor_i[intro!]: "constructor_i \<in> ObjSeqSet"
```

      ii. If a constructor has an associated precondition, it can be included in the declaration of the corresponding introduction rule.

4. Generate the skeleton containing the lemmas – *JMLmodel_spec.thy*

[7] This type of construct is called *record selector* in Isabelle.

(a) Generate for each method m_i the following piece of code. Some tactics to prove this lemmas may be already determined at this point. For simplicity's sake, the arguments are left out here.

i. `lemma post_m_i: "m_i ==> POST_m_i"`

ii. The lemmas may be additionally added to the set of simplification rules.

5. If $equational_theory_{model}$ exists then generate the corresponding specifications – *JMLmodel_eqTheory.thy*

   (a) For each conjunct generate an according lemma,

6. Generate the skeleton containing the invariants – *JMLmodel_invariants.thy*

   (a) For each $INV_i, i = 1, \ldots, k$ generate the following stub of code

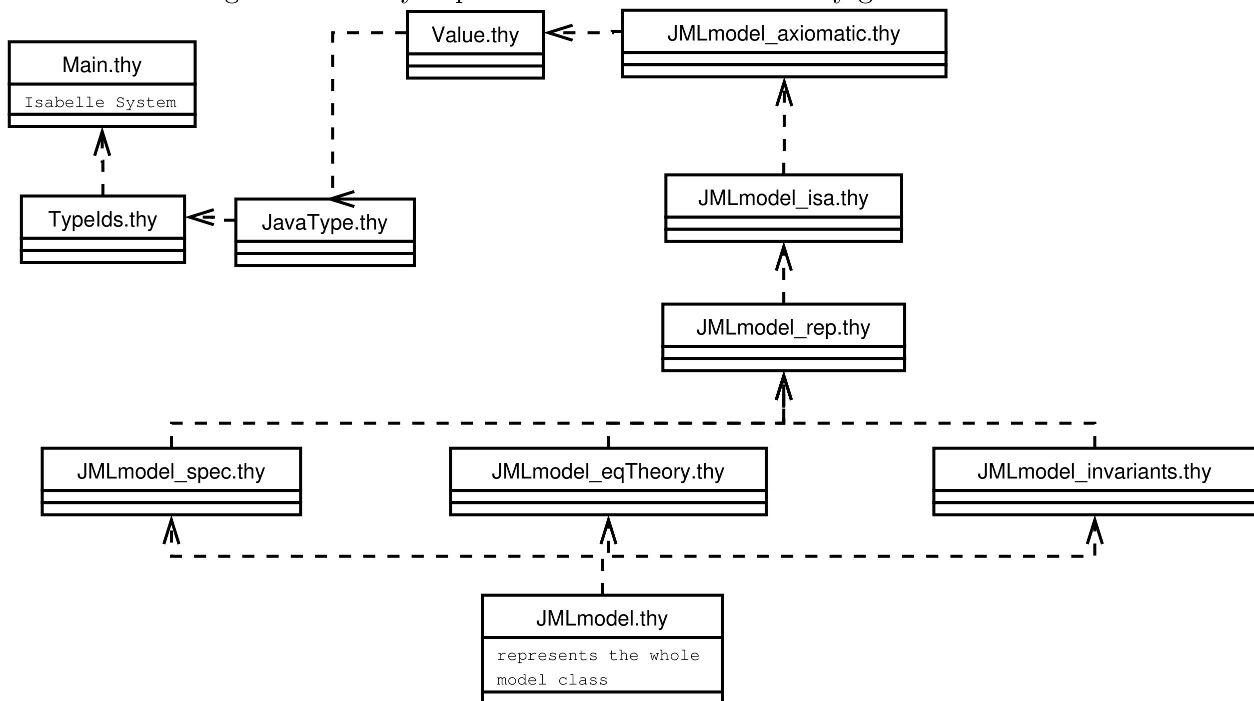
   i.
```
lemma inv_i: "o \in JMLmodelSet ==> INV_i"
apply(erule ObjSeqSet.induct)
apply(simp_all)
done
```

   ii. This sequence of proof commands can be left out, but during the experience with Isabelle and the class invariants they turned out to prove the lemma straight forward. Of course, some cases will be encountered where this proof fails.

4.4 Summary

The specifications/theories generated by this algorithm are graphically represented in the following figure

Figure 7: Theory dependencies of an automatically generated model



These dependencies were not explicitely stated in the algorithm above, but working with them in future could ease work, e.g. the representation of the model knows facts about the axiomatic definition, that could further be used.

Finally, we have found a way to generate a theory automatically. Clearly this generation is not complete and can be extended, if the structure of a model class allows it. Take *JMLMath* as an example:

due to its simplicity and the existence of static-only methods, our mapping would be complete, because the axiomatic specification would cover all aspects of the model class.

5 Conclusions

5.1 Achievements

An Adequate Mapping of two JML Model Classes , namely *JMLMath* and *JMLObjectSequence*. For *JMLObjectSequence* two different approaches were applied, one with simple lists and the other with inductive sets. Whereas the first mapping is rather simple, the second one should be preferred, as it better reflects the notion of objects and classes, and, moreover, is easier to extend.

A Tentative Definition of an Algorithm , that could generate such model classes automatically, has been given.

5.2 Further Work

There are several ways in which this work could be extended or applied:

Extension of the Model in the sense, that concepts can be refined, or that new concepts can be introduced, such as treatment of exceptional behaviour.

Investigation on further ways of doing such a mapping. One of this ways could be the use of *Axiomatic type classes*.

Implementation of the Algorithm , that generates the mapping of JML model classes. Possibly the JML-compiler *jmlc* will be of great help.

5.3 Final Statement

By modelling *JMLObjectSequence* and *JMLMath* in Isabelle/HOL it has been shown that

> mapping JML model classes in Isabelle/HOL is feasible, but it's difficult to find a way of generating them fully automatically.

May the result of this work, the experience as well as the techniques, be helpful for further research in this direction.

References

[1] Albert L. Baker Gary T. Leavens and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical report, 2003. Available from `http:\\www.jmlspecs.org`

[2] David von Oheimb Lawrence C. Paulson Thomas M. Rasmussen Christophe Tabacznyj Gertrud Bauer, Tobias Nipkow and Markus Wenzel. The supplemental Isabelle/HOL Library. 2002. Part of the Isabelle distribution `http://isabelle.in.tum.de/library/HOL/Library/document.pdf`

[3] C.A.R Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, October 1969.

[4] Gary T. Leavens and Yoonsik Cheon. Design by Contract with JML. 2004.

[5] Barbara Liskov and Jeanette Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, November 1994.

[6] Lawrence C. Paulson Tobias Nipkow and Markus Wenzel. *A Proof Assistant for Higher–Order Logic*, May 2003. Available from `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/`

[7] Markus Wenzel. *The Isabelle/Isar Reference Manual*. `http://isabelle.in.tum.de/doc/isar-ref.pdf`

A Figures

List of Figures

1 A class Person using JML annotations 8
2 A Theory of Lists 9
3 Proofs about lists 10
4 Theory dependency graph of the mapping with lists 17
5 Theory dependency graph of the mapping with inductive sets 24
6 Graphical description of a generic model 27
7 Theory dependencies of an automatically generated model 30

B Tables

List of Tables

1 JML extensions 6

C Isabelle Theories

C.1 JMLMath

JMLMath - all the definitions and lemmas together

```
theory JMLMath = Complex_Main:

types bigint = int

constdefs
floor         :: "real => real"
"floor r == real (IntFloor.floor r)"

ceil           :: "real => real"
"ceil r == real (ceiling r)"

nearestInteger :: "real => bigint"
"nearestInteger (a::real) == THE n::bigint.
                               abs (real n - a) <= ((1/2)::real) &
                              (abs(real n -a) = ((1/2)::real) --> n mod 2 = 0)"

rint           :: "real => real"
"rint a == real (nearestInteger a)"
round          :: "real => bigint"
"round a == IntFloor.floor (a + (1/2)::real)"

(* declare all constant definitions as simplification rules *)
declare floor_def          [simp]
declare ceil_def           [simp]
declare nearestInteger_def [simp]
declare rint_def           [simp]
declare round_def          [simp]

(* the postconditions *)

(*
  JMLspec bug. there should be a premise "0 <= a". If not there, sqrt will fail on negative numbers
*)
lemma post_sqrt_gt_0 [simp]: "0 <= a ==> 0 <= sqrt a"
apply(rule Transcendental.real_sqrt_ge_zero)
apply(auto)
done

(*
   again, JML spec has no pre-cond, while lemma does. Another JML bug.
*)
```

```
lemma post_sqrt_mult_equals [simp]: "0 <= a ==> sqrt a^2   = a"
by simp

lemma post_floor:"JMLMath.floor a <= a"
by simp

lemma post_ceil:"a \<le> ceil a"
by simp

lemma rint_eq_nearestInt [simp]: "real (nearestInteger a) = rint a"
by simp
```

C.2 JMLObjectSequence - Approach Primitive Lists

JMLObjectSequenceDef.thy - Definitions of types and functions

```
theory JMLObjectSequenceDef = Value:

(* helper function: converts an integer to a natural, if the integer is greater or equal to zero *)
constdefs
myNat:: "int => nat"
"0 \<le> i ==> myNat i \<equiv> nat i"

types JMLObjectSequence_rep = "value list"

(* Methods *)
consts
        count               :: "JMLObjectSequence_rep => value => int"
        containsAll         :: "JMLObjectSequence_rep => JMLObjectSequence_rep => bool"
        isPrefix            :: "JMLObjectSequence_rep => JMLObjectSequence_rep => bool"
        isProperPrefix      :: "JMLObjectSequence_rep => JMLObjectSequence_rep => bool"
        indexOf             :: "JMLObjectSequence_rep => value => int"
        trailer             :: "JMLObjectSequence_rep => JMLObjectSequence_rep"
        isSubsequence       :: "JMLObjectSequence_rep => JMLObjectSequence_rep => bool"
        isProperSubsequence :: "JMLObjectSequence_rep => JMLObjectSequence_rep => bool"
        isInsertionInto     :: "JMLObjectSequence_rep => JMLObjectSequence_rep => value => bool"

primrec
"count [] x = 0"
"count (h#t) x = (if h=x then 1 + (count t x) else count t x)"

primrec
"containsAll [] c = True"
"containsAll (h#t) c = (if (h mem c) then (containsAll t c) else False)"

primrec
"isPrefix xs [] = (case xs of [] => True | z#zs => False)"
"isPrefix xs (y#ys) = (case xs of [] => True | z#zs => (if (z=y) then (isPrefix zs ys) else False))"

primrec
"isProperPrefix xs [] = False"
"isProperPrefix xs (y#ys) = (case xs of [] => True | z#zs => (if (z=y) then (isPrefix zs ys) else False))"

primrec
"indexOf [] obj  = -1"
"indexOf (h#t) obj = (if h=obj then 0 else 1 + (indexOf t obj))"

primrec
"trailer [] = []"
"trailer (h#t) = t"

primrec
"isSubsequence xs [] = False"
"isSubsequence xs (h#t) = (if (isPrefix xs (h#t)) then True else (isSubsequence xs t))"

primrec
"isProperSubsequence xs [] = False"
"isProperSubsequence xs (h#t) = (if (isProperPrefix xs (h#t)) then True else (isProperSubsequence xs t))"

primrec
```

```
"isInsertionInto xs [] ys = ((hd(xs)=ys) & (length xs)=1)"
"isInsertionInto xs (h#t) ys = (case xs of [] => False
                               | h1#t1 => (if h1=ys then ((isPrefix t1 t) & (containsAll t1 t))
                                           else (if h1=h then (isInsertionInto t1 t ys)
                                                 else False
                                                )
                                           )
                               )"

(* functions that can be defined using other functions *)
constdefs
(* corresponds to model field EMPTY *)
EMPTY :: "JMLObjectSequence_rep"
"EMPTY == []"

singleton :: "value => JMLObjectSequence_rep"
"singleton xs == xs#[]"

convertFrom :: "JMLObjectSequence_rep => JMLObjectSequence_rep"
"convertFrom x == x"

convertFromWithSize :: "JMLObjectSequence_rep => int => JMLObjectSequence_rep"
"convertFromWithSize x i == take (myNat i) x"

itemAt :: "JMLObjectSequence_rep => int => value"
"itemAt l i == nth l (myNat i)"

get :: "JMLObjectSequence_rep => int => value"
"get j i == itemAt j i"

has :: "JMLObjectSequence_rep => value => bool"
"has l x == x mem l"

isSuffix :: "JMLObjectSequence_rep => JMLObjectSequence_rep => bool"
"isSuffix s1 s2 == isPrefix (rev s1) (rev s2)"

isProperSuffix :: "JMLObjectSequence_rep => JMLObjectSequence_rep => bool"
"isProperSuffix s1 s2 == isProperPrefix (rev s1) (rev s2)"

isSupersequence :: "JMLObjectSequence_rep => JMLObjectSequence_rep => bool"
"isSupersequence xs xt == isSubsequence xt xs"

isProperSupersequence :: "JMLObjectSequence_rep => JMLObjectSequence_rep => bool"
"isProperSupersequence xs xt == isProperSubsequence xt xs"

isDeletionFrom::"JMLObjectSequence_rep => JMLObjectSequence_rep => value => bool"
"isDeletionFrom l1 l2 xs == isInsertionInto l2 l1 xs"

isEmpty :: "JMLObjectSequence_rep => bool"
"isEmpty l == null l"

first :: "JMLObjectSequence_rep => value"
"first l == (hd l)"

prefix :: "JMLObjectSequence_rep => int => JMLObjectSequence_rep"
"prefix l n == take (myNat n) l"

removePrefix :: "JMLObjectSequence_rep => int => JMLObjectSequence_rep"
"removePrefix l n == drop (myNat n) l"

concat :: "JMLObjectSequence_rep => JMLObjectSequence_rep => JMLObjectSequence_rep"
"concat t s == (t @ s)"

reverse :: "JMLObjectSequence_rep => JMLObjectSequence_rep"
"reverse s == (rev s)"

headerF :: "JMLObjectSequence_rep => JMLObjectSequence_rep"
"headerF l == (butlast l)"

insertFront :: "JMLObjectSequence_rep => value => JMLObjectSequence_rep"
"insertFront l x == (x#[]) @ l"

insertBack :: "JMLObjectSequence_rep => value => JMLObjectSequence_rep"
```

```
"insertBack l x == l @ (x#[])"

removeItemAt :: "JMLObjectSequence_rep => int => JMLObjectSequence_rep"
"removeItemAt x i == (take (myNat i) x) @ (drop ((myNat i) + 1) x)"

replaceItemAt::"JMLObjectSequence_rep => int => value => JMLObjectSequence_rep"
"replaceItemAt x i xs == (take (myNat i) x) @ (xs#[]) @ (drop ((myNat i) + 1) x)"

insertAfterIndex :: "JMLObjectSequence_rep => int => value => JMLObjectSequence_rep"
"insertAfterIndex x i xs == (take ((myNat i) + 1) x) @ (xs#[]) @ (drop ((myNat i) + 1) x)"

insertBeforeIndex :: "JMLObjectSequence_rep => int => value => JMLObjectSequence_rep"
"insertBeforeIndex x i xs == (take ((myNat i) - 1) x) @ (xs#[]) @ (drop ((myNat i) - 1) x)"

subsequence :: "JMLObjectSequence_rep => int => int => JMLObjectSequence_rep"
"subsequence x i j == take (myNat (i-j+1)) (drop (myNat i) x)"

(* should be JMLObjectSequence_rep => new *)
clone :: "JMLObjectSequence_rep => JMLObjectSequence_rep"
"clone x == x"

(* should be JMLObjectSequence_rep => value => bool *)
equals::"JMLObjectSequence_rep => JMLObjectSequence_rep => bool"
"equals s1 s2 == isPrefix s1 s2 & isSuffix s2 s1"
(* constructors *)
JMLObjectSequence_cons1 :: "JMLObjectSequence_rep"
"JMLObjectSequence_cons1 == EMPTY"

JMLObjectSequence_cons2 :: "value => JMLObjectSequence_rep"
"JMLObjectSequence_cons2 xs == singleton xs"

(* declare all constant definitions as simplification rules *)
declare myNat_def               [simp]
declare EMPTY_def               [simp]
declare singleton_def           [simp]
declare convertFrom_def         [simp]
declare convertFromWithSize_def [simp]
declare itemAt_def              [simp]
declare get_def                 [simp]
declare has_def                 [simp]
declare isSuffix_def            [simp]
declare isProperSuffix_def      [simp]
declare isSupersequence_def     [simp]
declare isProperSupersequence_def[simp]
declare isDeletionFrom_def      [simp]
declare isEmpty_def             [simp]
declare first_def               [simp]
declare prefix_def              [simp]
declare removePrefix_def        [simp]
declare concat_def              [simp]
declare reverse_def             [simp]
declare headerF_def             [simp]
declare insertFront_def         [simp]
declare insertBack_def          [simp]
declare removeItemAt_def        [simp]
declare replaceItemAt_def       [simp]
declare insertAfterIndex_def    [simp]
declare insertBeforeIndex_def   [simp]
declare subsequence_def         [simp]
declare clone_def               [simp]
declare equals_def              [simp]
declare JMLObjectSequence_cons1_def [simp]
declare JMLObjectSequence_cons2_def [simp]

end
```

JMLObjectSequence.thy - pre- and postconditions

```
theory JMLObjectSequence = JMLObjectSequenceDef:

lemma post_singleton: "singleton e \<noteq> [] & singleton e = JMLObjectSequence_cons2 e"
by simp

lemma post_convertFrom:" size (convertFrom a) = length a
```

```
                                              & (
                                                    \<forall> (i::int). 0 <= i & i < int (length a) -->
                                                    (
                                                     if (itemAt (convertFrom a) i) = Null then
                                                         (itemAt a i) = Null
                                                     else (itemAt (convertFrom a) i) = (itemAt a i)
                                                     )

                                               )"
by simp

(*
lemma post_convertFromWithSize:"0 <= (j::int) & j <= int (length a) ==>
                                              size (convertFromWithSize a j) = length a
                                              & (
                                                    \<forall> (i::int). 0 <= i & i < int (length a) -->
                                                    (
                                                     if (itemAt (convertFromWithSize a j) i) = Null then
                                                         (itemAt a i) = Null
                                                     else (itemAt (convertFromWithSize a j) i) = (itemAt a i)
                                                     )
                                                )"
apply (simp)
apply (auto)
apply (induct_tac a)
apply (auto)
done
*)

lemma post_size:"size xs == length xs"
by simp

lemma post_length:"length xs == size xs"
by simp

lemma post_impl_count :"0 \<le> (count x i)"
apply(induct_tac x)
by auto

(*
lemma post_has_1:"elem \<noteq> Null ==>
(has x elem) = (\<exists> i::int. 0 <= i & i < int (length x) & (itemAt x i) = elem)"
apply(simp)
apply(auto)
done
*)

(*
lemma post_has_2:"elem = Null ==>
has x elem --> (\<exists> i::int. 0 <= i & i < int (length x) & (itemAt x i) = Null)"
done
*)

(*
lemma post_isPrefix: "(isPrefix this s2) =
((length this <= length s2) &
              (\<forall> i::int. 0 <= i & (i < int (length this)) &
              ((itemAt s2 i) \<noteq> Null & (itemAt s2 i) = (itemAt this i))
               | ( (itemAt s2 i) = Null & (itemAt this i)= Null )))"
apply(simp)
done
*)

(*
lemma post_isProperPrefix: "isProperPrefix this s2  = (isPrefix this s2 & ~ equals this s2)"
apply(simp)
apply(auto)
apply(case_tac this)
apply(auto)
apply(case_tac s2)
apply(auto)
done
*)
```

```
(*
lemma post_isSuffix: "(isSuffix this s2) =
(((length this) <= (length s2) & (equals this (removePrefix s2 (int (length s2 - length this))))))"
apply(simp)
apply(auto)
done
*)

(*
lemma post_isProperSuffix: "isProperSuffix this s2 = ((isSuffix this s2) & ~(equals this s2)"
done
*)

lemma post_isEmpty:"isEmpty s = (length s = 0)"
apply(simp)
apply(induct_tac s)
apply(auto)
done

lemma post_indexOf_1:"item \<noteq> Null ==>
 itemAt this (indexOf this item) = item & (\<forall> (i::int). 0 <= i & i < (indexOf this item)
& ~((itemAt this i) = item))"
apply(simp)
apply(auto)
done
*)

(*
lemma post_indexOf_2:"item = Null ==>
 (itemAt this (indexOf this item)) = Null
 & ( \<forall> (i::int). 0 <= i & i < (indexOf this item) & (itemAt this i) \<noteq> Null)"
done
*)

lemma post_first_1:"0 < length this & (itemAt this 0) \<noteq> Null ==> (first this) = itemAt this 0"
apply(case_tac this)
by auto

lemma post_first_2:"0 < length this & (itemAt this 0) = Null ==> (first this) = Null"
apply(case_tac this)
by auto

(*
lemma post_last_1:"1 \<le> length this & (itemAt this (int(length this) - 1)) \<noteq> Null ==>
                                     last this = (itemAt this (int(length this) - 1))"
apply(case_tac this)
apply(auto)
done
*)

(*
lemma post_last_2:"1 \<le> length this & (itemAt this (int(length this) - 1)) = Null ==>
                                     last this = Null"
apply(case_tac this)
apply(auto)
done
*)

(*
lemma post_isSubsequence: "isSubsequence this s2 = ((length this) \<le> (length s2) &
(\<exists> (i::int). 0 \<le> i & i < int ((length s2) - (length this) + 1) &
(isPrefix this (removePrefix s2 i))))"
apply(auto)
done
*)

(*
lemma post_isProperSubsequence: "(isProperSubsequence this s2) = ((isSubsequence this s2) & ~(equals this s2))"
apply(simp)
apply(auto)
done
*)
```

```
lemma post_isSupersequence: "(isSupersequence this s2) = (isSubsequence s2 this)"
by simp

lemma post_isProperSupersequence: "(isProperSupersequence this s2) = (isProperSubsequence s2 this)"
by simp

(*
lemma post_isInsertionInto: "(isInsertionInto this s2 elem) =
(\<exists> (i::int). 0 \<le> i & i < int (length this) & (itemAt this i) = elem &
equals (concat (subsequence this 0 i) (subsequence this (i + 1) (int(length this)))) s2)"
apply(simp)
apply(auto)
done
*)

(*
lemma post_isInsertionInto_redund_1: "(isInsertionInto this s2 elem) ==> (length this)= (length s2)+1"
apply(simp)
apply(case_tac this)
apply(auto)
done
*)

(*
lemma post_isInsertionInto_redund_2: "(isInsertionInto this s2 elem) ==> (has this elem)"
apply(simp)
apply(auto)
done
*)

lemma post_isInsertionInto_redund_3: "(isInsertionInto this s2 elem) = (isDeletionFrom s2 this elem)"
by simp

(*
lemma post_isDeletionFrom: "(isDeletionFrom this s2 elem) =
(\<exists> (i::int). 0 \<le> i & i < int(length this) &
(itemAt this i) = elem & equals this (removeItemAt s2 i))"
apply(simp)
done
*)

(*
lemma post_isDeletionFrom_redund_1: "(isDeletionFrom this s2 elem)==> (length this)+1 = (length s2)"
apply(simp)
apply(auto)
done
*)

(*
lemma post_isDeletionFrom_redund_2: "(isDeletionFrom this s2 elem) ==> (has s2 elem)"
apply(simp)
done
*)

lemma post_isDeletionFrom_redund_3: "(isDeletionFrom this s2 elem) = (isInsertionInto s2 this elem)"
by simp

(* only postcond we can model for clone *)
lemma post_clone:"(clone this) = this"
by simp

(*
lemma post_prefix: "0 \<le> (n::int) & n \<le> int(length this) ==>
int (length (prefix this n)) = n & (\<forall> (i::int). 0 \<le> i & i < n &
(
 ( (itemAt (prefix this n) i) \<noteq> Null --> (itemAt (prefix this n) i) = (itemAt this i) )
|
 ( (itemAt (prefix this n) i) = Null --> (itemAt this i) = Null )
))"
apply(simp)
apply(case_tac this)
apply(auto)
```

```
done
*)

(*
lemma post_removePrefix: "0 \<le> (n::int) & n <= int (length this) ==>
int(length (prefix this n)) = int(length this) - n & (\<forall> (i::int). 0 \<le> i & i < int (length this) &
 (
  ((itemAt (removePrefix this n) (i-n)) \<noteq> Null & (itemAt (removePrefix this n) (i-n)) = (itemAt this i))
 |
  ((itemAt (removePrefix this n) (i-n)) = Null & (itemAt this i) = Null )))"
done
*)

(*
lemma post_concat: "length (concat this s2) = (length this) + (length s2)
& (\<forall> (i::int). 0 \<le> i & i < int (length this)
& (
   ( (itemAt (concat this s2) i) \<noteq> Null & (itemAt (concat this s2) i) = (itemAt this i) )
  |
   ( (itemAt (concat this s2) i) = Null & (itemAt this i) = Null )
  ))
& (\<forall> (i::int). 0 \<le> i & i < int (length s2)
& (
  ( (itemAt (concat this s2) (int (length his) + i)) \<noteq> Null
   & (itemAt (concat this s2) (int (length his) + i)) = (itemAt s2 i) )
 |
  ( (itemAt (concat this s2) (int (length his) + i)) = Null & (itemAt s2 i) = Null )
 ))"
apply(simp)
apply(auto)
done
*)

(*
lemma post_reverse: "length this = length (reverse this) &
(\<forall> (i::int). 0 \<le> i & i \<le> int (length this)
 & (
   ( itemAt (reverse this) (int (length this) - i - 1) \<noteq> Null
   & itemAt (reverse this) (int (length this) - i - 1) = (itemAt this i)
   )
 |
 ( itemAt (reverse this) (int (length this) - i - 1) = Null
 & itemAt this i = Null
 )))"
apply(simp)
done
*)

(*
lemma post_removeItemAt: "0 \<le> (index::int) & index < int (length this) ==>
equals (removeItemAt this index) (concat (prefix this index) (removePrefix this (index+1)))"
apply(simp)
apply(case_tac this)
apply(auto)
done
*)

(*
lemma post_replaceItemAt: "0 \<le> (index::int) & index < int (length this) ==>
equals (replaceItemAt this index item) (inserBeforeIndex (removeItemAt this index) index item)"
apply(simp)
done
*)

(*
lemma post_header: "1 \<le> length this ==> equals (headerF this) (removeItemAt this (int (length this) - 1))"
apply(simp)
apply(case_tac this)
done
*)

lemma post_header_redund: "1 \<le> length this ==> length (headerF this) = ((length this) - 1)"
by simp
```

```
(*
lemma post_trailer: " 1 \<le> length this ==> equals (trailer this) (removePrefix this 1)"
apply(simp)
apply(case_tac this)
done
*)

lemma post_trailer_redund: "1 \<le> length this ==> length (trailer this) = ((length this) - 1)"
apply(simp)
apply(case_tac this)
apply(auto)
done

(*
lemma post_insertAfterIndex: " -1 <= (i::int) & i < int (length this) ==>
equals (insertAfterIndex this i item) (insertBeforeIndex this (i + 1) item)"
apply(simp)
done
*)

(*
lemma post_insertBeforeIndex: "0 \<le> i & i \<le> int (length this) & (length this) < MAX_VALUE ==>
                                              equals (insertBeforeIndex this i item)
                                                     (
                                                     concat (
                                                       concat (prefix this i) (JMLObjectSequence_cons2 item)
                                                           )
                                                            (removePrefix this i))"
done
*)

(*
lemma post_insertBack: "int (length this) < MAX_VALUE ==>
equals (insertBack this item) (insertBeforeIndex this (int(length this)) item)"
done

lemma post_insertBack_redund_1: "int (length this) < MAX_VALUE ==>
length(insertBack this item) = (length this) + 1 & (isProperPrefix this (insertBack this item))"
done
*)

(*
lemma post_insertFront: "int (length this) < MAX_VALUE ==>
equals (insertFront this item) (insertBeforeIndex this 0 item)"
done

lemma post_insertFront_redund_1: "int (length this) < MAX_VALUE ==>
length(insertFront this item) = (length this) + 1 & equals (trailer (insertFront this item)) this"
done
*)

(*
lemma post_subsequence: "0 \<le> (i::int) & i \<le> (j::int) & j \<le> int (length this) ==>
equals (subsequence this i j) (prefix (removePrefix this j) (j - i))"
apply(simp)
apply(auto)
done
*)

(*
lemma post_subsequence_redund: "int (length (subsequence this i j)) = (i - j)"
apply(simp)
done
*)

end
```

JMLObjectSequenceEquationalTheory.thy - lemmas concerning the equational theory

```
theory JMLObjectSequenceEquationalTheory = JMLObjectSequence:

lemma eq_theory_1: "\<not> has JMLObjectSequence_cons1 e1"
by simp

lemma eq_theory_2: "size JMLObjectSequence_cons1 = 0"
by simp

lemma eq_theory_3: "size (JMLObjectSequence_cons2 e1) = 1"
by simp

lemma eq_theory_4: "e1 \<noteq> Null ==> itemAt (JMLObjectSequence_cons2 e1) 0 = e1"
by simp

lemma eq_theory_5: "e1 \<noteq> Null ==> first (insertFront s e1) = e1"
by simp

lemma eq_theory_6:"e1 \<noteq> Null ==> size (insertFront s e1) == size s + 1"
by simp

lemma eq_theory_7: "e1 \<noteq> Null ==> last (insertBack s e1) = e1"
by simp

lemma eq_theory_8: "e1 \<noteq> Null ==> itemAt (insertBack s e1) (int (size s)) = e1"
by simp


(* Taken from specification
// !FIXME! The following may be inconsistent: argument to itemAt might
// be equal to size, but it is required to be less than size.

lemma eq_theory_9: "-1 <= n & n < int (size s) & e1 \<noteq> Null  ==>
itemAt (insertAfterIndex s n e1) (n+1) = e1"
apply(simp)
apply(auto)
*)


lemma eq_theory_10: "-1 <= n & n < int (size s) ==> size (insertAfterIndex s n e1) = (size s) + 1"
apply(simp)
by arith

(*
lemma eq_theory_11: "0 <= n & n <= int (size s) & e1 \<noteq> Null  ==>
itemAt (insertBeforeIndex s n e1) n = e1"
apply(simp)
apply(auto)
*)

lemma eq_theory_12: " 0 <= n & n <= int (size s) ==> size (insertBeforeIndex s n e1) = (size s) + 1"
apply(simp)
apply(auto)
by arith

(*
lemma eq_theory_13: "(isPrefix s s2) = (\<forall> i. 0 <= i & i < int (length s) &
                                        ( (itemAt s i) \<noteq> Null & (itemAt s2 i) = (itemAt s i))
                                        |
                                        ( (itemAt s2 i) = Null & (itemAt s i) = Null)
                                        )"
apply(simp)
apply(auto)
*)

(*
lemma eq_theory_14: "(isSubsequence s s2) = ( length s <= length s2 & ( (isPrefix s s2) |
                                            (isSubsequence s (trailer s2))) )"
apply(auto)
*)

lemma eq_theory_15: "(isEmpty s) = (size s = 0)"
apply(simp)
```

```
apply(case_tac s)
by auto

lemma eq_theory_16: "(isEmpty s) = (length s = 0)"
apply(simp)
apply(case_tac s)
by auto

lemma eq_theory_17: "equals (JMLObjectSequence_cons2 e1) (insertFront JMLObjectSequence_cons1 e1)"
by simp

lemma eq_theory_18: "equals (JMLObjectSequence_cons2 e1) (insertBack JMLObjectSequence_cons1 e1)"
by simp

lemma eq_theory_19: "equals (JMLObjectSequence_cons2 e1) (insertAfterIndex JMLObjectSequence_cons1 -1 e1)"
by simp

lemma eq_theory_20: "equals (JMLObjectSequence_cons2 e1) (insertBeforeIndex JMLObjectSequence_cons1 0 e1)"
by simp

(* helper lemma for eq_theory_21 *)
lemma isPrefix_eq [simp]:"isPrefix s s"
apply(induct_tac s)
apply(auto)
done

lemma eq_theory_21: "1 \<le> (size s) ==> equals s (insertFront (trailer s) (first s))"
apply(simp)
apply(case_tac s)
by auto


(*
lemma helper_22_1 [simp]:"isPrefix s (butlast s @ (last s)#[])"
apply(induct_tac s)
apply(auto)
done

lemma helper_22_2 [simp]:"1 \<le> length s ==> isPrefix ((last s)#[] @ (rev (butlast s))) (rev s)"
apply(simp)
apply(case_tac s)
apply(auto)


lemma eq_theory_22: "1 \<le> (size s) ==> equals s (insertBack (headerF s) (last s))"
apply(simp)
apply(auto)
by
*)

(*
lemma eq_theory_23: "(isProperSubsequence s s2) = ((isSubsequence s s2) & (\<not>(equals s s2)))"
apply(simp)
*)

lemma eq_theory_24: "isSupersequence s s2 = isSubsequence s2 s"
by simp

lemma eq_theory_25: "isProperSupersequence s s2 = isProperSubsequence s2 s"
by simp

end
```

C.3 JMLObjectSequence - Approach Inductive Sets

JMLObjectSequence.thy - theory summarizing all specifications

```
theory JMLObjectSequence  = JMLObjectSequence_invariants + JMLObjectSequence_spec + JMLObjectSequence_eqTheory:

end
```

JMLObjectSequence_isa.thy - the Isabelle representation

```
theory JMLObjectSequence_isa = Value:

types JMLObjectSequence_isa = "value list"

constdefs
myNat:: "int => nat"
"0 \<le> i ==> myNat i \<equiv> nat i"

(* Methods *)
consts
        count               :: "JMLObjectSequence_isa => value => int"
        containsAll         :: "JMLObjectSequence_isa => JMLObjectSequence_isa => bool"
        isPrefix            :: "JMLObjectSequence_isa => JMLObjectSequence_isa => bool"
        isProperPrefix      :: "JMLObjectSequence_isa => JMLObjectSequence_isa => bool"
        indexOf             :: "JMLObjectSequence_isa => value => int"
        isSubsequence       :: "JMLObjectSequence_isa => JMLObjectSequence_isa => bool"
        isProperSubsequence :: "JMLObjectSequence_isa => JMLObjectSequence_isa => bool"
        isInsertionInto     :: "JMLObjectSequence_isa => JMLObjectSequence_isa => value => bool"

primrec
"count [] x = 0"
"count (h#t) x = (if h=x then 1 + (count t x) else count t x)"

primrec
"containsAll [] c = True"
"containsAll (h#t) c = (if (h mem c) then (containsAll t c) else False)"

primrec
"isPrefix xs [] = (case xs of [] => True | z#zs => False)"
"isPrefix xs (y#ys) = (case xs of [] => True | z#zs => (if (z=y) then (isPrefix zs ys) else False))"

primrec
"isProperPrefix xs [] = False"
"isProperPrefix xs (y#ys) = (case xs of [] => True | z#zs => (if (z=y) then (isPrefix zs ys) else False))"

primrec
"indexOf [] obj  = -1"
"indexOf (h#t) obj = (if h=obj then 0 else 1 + (indexOf t obj))"

primrec
"isSubsequence xs [] = False"
"isSubsequence xs (h#t) = (if (isPrefix xs (h#t)) then True else (isSubsequence xs t))"

primrec
"isProperSubsequence xs [] = False"
"isProperSubsequence xs (h#t) = (if (isProperPrefix xs (h#t)) then True else (isProperSubsequence xs t))"

primrec
"isInsertionInto xs [] ys = ((hd(xs)=ys) & (length xs)=1)"
"isInsertionInto xs (h#t) ys = (case xs of [] => False
  | h1#t1 =>
  (if h1=ys then ((isPrefix t1 t) & (containsAll t1 t))
   else (if h1=h then (isInsertionInto t1 t ys) else False)))"

(* functions that can be defined using other functions *)
constdefs
itemAt :: "JMLObjectSequence_isa => int => value"
"itemAt l i == nth l (myNat i)"
get :: "JMLObjectSequence_isa => int => value"
"get j i == itemAt j i"
has :: "JMLObjectSequence_isa => value => bool"
"has l x == x mem l"
isSuffix :: "JMLObjectSequence_isa => JMLObjectSequence_isa => bool"
"isSuffix s1 s2 == isPrefix (rev s1) (rev s2)"
isProperSuffix :: "JMLObjectSequence_isa => JMLObjectSequence_isa => bool"
```

```
"isProperSuffix s1 s2 == isProperPrefix (rev s1) (rev s2)"
isSupersequence :: "JMLObjectSequence_isa => JMLObjectSequence_isa => bool"
"isSupersequence xs xt == isSubsequence xt xs"
isProperSupersequence :: "JMLObjectSequence_isa => JMLObjectSequence_isa => bool"
"isProperSupersequence xs xt == isProperSubsequence xt xs"
isDeletionFrom::"JMLObjectSequence_isa => JMLObjectSequence_isa => value => bool"
"isDeletionFrom l1 l2 xs == isInsertionInto l2 l1 xs"
isEmpty :: "JMLObjectSequence_isa => bool"
"isEmpty l == null l"
first :: "JMLObjectSequence_isa => value"
"first l == (hd l)"
equals::"JMLObjectSequence_isa => JMLObjectSequence_isa => bool"
"equals s1 s2 == isPrefix s1 s2 & isSuffix s2 s1"
headerF :: "JMLObjectSequence_isa => JMLObjectSequence_isa"
"headerF s \<equiv> butlast s"

(* functions returning isabelle-representation of the JMLObjectSequence *)
consts
trailer_isa :: "JMLObjectSequence_isa => JMLObjectSequence_isa"

primrec
"trailer_isa []    = []"
"trailer_isa (h#t) = t"

constdefs
(* returns an empty isa-Sequence *)
EMPTY_isa :: "JMLObjectSequence_isa"
"EMPTY_isa \<equiv> []"

(* returns the trailer
trailer_isa :: "JMLObjectSequence_isa => JMLObjectSequence_isa"
"trailer_isa s1 \<equiv> t"
*)

(* returns an isa-Sequence with just one element *)
singleton_isa :: "value => value list"
"singleton_isa xs == xs#[]"

(* concatenates two isa-sequences *)
concat_isa :: "JMLObjectSequence_isa => JMLObjectSequence_isa => JMLObjectSequence_isa"
"concat_isa t s \<equiv> (t @ s)"

insertAfterIndex_isa :: "JMLObjectSequence_isa => int => value => JMLObjectSequence_isa"
"insertAfterIndex_isa x i xs \<equiv> (take ((nat i) + 1) x) @ (xs#[]) @ (drop ((nat i) + 1) x)"

insertBeforeIndex_isa :: "JMLObjectSequence_isa => int => value => JMLObjectSequence_isa"
"insertBeforeIndex_isa x i xs == (take ((nat i) - 1) x) @ (xs#[]) @ (drop ((nat i) - 1) x)"

prefix_isa :: "JMLObjectSequence_isa => int => JMLObjectSequence_isa"
"prefix_isa l n == take (nat n) l"

replaceItemAt_isa :: "value list => int => value => value list"
"replaceItemAt_isa x i xs == (take (nat i) x) @ (xs#[]) @ (drop ((nat i) + 1) x)"

removeItemAt_isa :: "JMLObjectSequence_isa => int => JMLObjectSequence_isa"
"removeItemAt_isa x i == (take (nat i) x) @ (drop ((nat i) + 1) x)"

removePrefix_isa :: "JMLObjectSequence_isa => int => JMLObjectSequence_isa"
"removePrefix_isa l n == drop (nat n) l"

subsequence_isa :: "JMLObjectSequence_isa => int => int => JMLObjectSequence_isa"
"subsequence_isa x i j == take (nat (i-j+1)) (drop (nat i) x)"

insertFront_isa :: "JMLObjectSequence_isa => value => JMLObjectSequence_isa"
"insertFront_isa l x == (x#[]) @ l"

declare myNat_def               [simp]
declare itemAt_def              [simp]
declare get_def                 [simp]
declare has_def                 [simp]
declare isSuffix_def            [simp]
declare isProperSuffix_def      [simp]
declare isSupersequence_def     [simp]
```

```
declare isProperSupersequence_def[simp]
declare isDeletionFrom_def      [simp]
declare isEmpty_def             [simp]
declare first_def               [simp]
declare equals_def              [simp]
declare headerF_def             [simp]
declare EMPTY_isa_def                 [simp]
declare concat_isa_def                [simp]
declare insertAfterIndex_isa_def      [simp]
declare insertBeforeIndex_isa_def     [simp]
declare prefix_isa_def                [simp]
declare removeItemAt_isa_def          [simp]
declare removePrefix_isa_def          [simp]
declare replaceItemAt_isa_def         [simp]
declare singleton_isa_def             [simp]
declare subsequence_isa_def           [simp]
declare insertFront_isa_def     [simp]
end
```

JMLObjectSequence_rep.thy - actual representation as tuple the inductive set

```
theory JMLObjectSequence_rep = JMLObjectSequence_isa:

(* type declarations *)

(* declare some types as given whithout specifying a
   precise definition
*)
types Class = nat
types JMLDataGroup = nat
types String = nat

(* model fields, inherited from java.lang.Object *)
record ModelFields = getClass   :: Class
                     objectState :: JMLDataGroup
                     theString   :: String

(* ghost fields, inherited from java.lang.Object *)
record GhostFieldsAndModelFields = ModelFields +
                     objectTimesFinalized :: int
                     owner                :: value (* should be of type object *)
(* ghost fields, inherited from org.jmlspecs.model.JMLCollection *)
                     containsNull         :: bool
                     elementType          :: javatype

(* class fields, just the important ones, the Strings used for exception-handling
   are not included
*)
record JMLObjectSequence_rep = GhostFieldsAndModelFields +
                     theLength :: int
                     theSeq    :: JMLObjectSequence_isa (*should be JMLObjectNodeList *)

constdefs
JMLObjectSequence_stdcons :: JMLObjectSequence_rep
"JMLObjectSequence_stdcons \<equiv> (| getClass = 0,
                                   objectState = 0,
                                   theString = 0,
                                   objectTimesFinalized = 0,
                                   owner = Null,
                                   containsNull = False,
                                   elementType = ClassT Object,
                                   theLength = 0,
                                   theSeq = EMPTY_isa
                                  |)"

JMLObjectSequence_cons2 :: "value => JMLObjectSequence_rep"
"JMLObjectSequence_cons2 e \<equiv> (| getClass = 0,
                                   objectState = 0,
                                   theString = 0,
                                   objectTimesFinalized = 0,
                                   owner = Null,
                                   containsNull = (typeof e = NullT),
```

```
                                   elementType = (if (typeof e = NullT) then (ClassT Object)
                                                  else (typeof e)),
                                   theLength = 1,
                                   theSeq = singleton_isa e
                                  |)"

(* FIX: definition of containsNull and elementtype *)
(* Suppose, that jmlspec is not right... see containsNull *)
JMLObjectSequence_cons3 :: "JMLObjectSequence_isa => int => JMLObjectSequence_rep"
"JMLObjectSequence_cons3 isaSeq i \<equiv> (| getClass = 0,
                                   objectState = 0,
                                   theString = 0,
                                   objectTimesFinalized = 0,
                                   owner = Null,
                                   containsNull = (if (isaSeq = []) then False else True),
                                   elementType = ClassT Object,
                                   theLength = i,
                                   theSeq = isaSeq
                                  |)"

(* other methods returning JMLObjectSequences *)
concat                 :: "JMLObjectSequence_rep => JMLObjectSequence_rep => JMLObjectSequence_rep"
"concat s1 s2 \<equiv>
JMLObjectSequence_cons3 (concat_isa (theSeq s1) (theSeq s2)) (theLength s1 + theLength s2)"

(* convertFrom Methods left out at the moment *)

headerSeq              :: "JMLObjectSequence_rep => JMLObjectSequence_rep"
"headerSeq s1 \<equiv> JMLObjectSequence_cons3 (butlast (theSeq s1)) ((theLength s1) - 1)"

insertAfterIndex       :: "JMLObjectSequence_rep => int => value => JMLObjectSequence_rep"
"insertAfterIndex s1 afterThisOne item \<equiv>
JMLObjectSequence_cons3 (insertAfterIndex_isa (theSeq s1) afterThisOne item) ((theLength s1) + 1)"

insertBack             :: "JMLObjectSequence_rep => value => JMLObjectSequence_rep"
"insertBack s1 item \<equiv> JMLObjectSequence_cons3 ((theSeq s1) @ (item#[])) ((theLength s1) + 1)"

insertBeforeIndex      :: "JMLObjectSequence_rep => int => value => JMLObjectSequence_rep"
"insertBeforeIndex s1 beforeThisOne item \<equiv>
JMLObjectSequence_cons3 (insertBeforeIndex_isa (theSeq s1) beforeThisOne item) ((theLength s1) + 1)"

insertFront            :: "JMLObjectSequence_rep => value => JMLObjectSequence_rep"
"insertFront s1 item \<equiv> JMLObjectSequence_cons3 ((item#[]) @ (theSeq s1)) ((theLength s1) + 1)"

prefix                 :: "JMLObjectSequence_rep => int => JMLObjectSequence_rep"
"prefix s1 i \<equiv> JMLObjectSequence_cons3 (prefix_isa (theSeq s1) (i)) (i)"

removeItemAt           :: "JMLObjectSequence_rep => int => JMLObjectSequence_rep"
"removeItemAt s1 i \<equiv> JMLObjectSequence_cons3 (removeItemAt_isa (theSeq s1) (i)) ((theLength s1) - 1)"

removePrefix           :: "JMLObjectSequence_rep => int => JMLObjectSequence_rep"
"removePrefix s1 i \<equiv> JMLObjectSequence_cons3 (removePrefix_isa (theSeq s1) (i)) ((theLength s1) - i)"

replaceItemAt          :: "JMLObjectSequence_rep => int => value => JMLObjectSequence_rep"
"replaceItemAt s1 i item \<equiv>
JMLObjectSequence_cons3 (replaceItemAt_isa (theSeq s1) i item) (theLength s1)"

singleton              :: "value => JMLObjectSequence_rep"
"singleton e \<equiv> JMLObjectSequence_cons2 e"

subsequence            :: "JMLObjectSequence_rep => int => int => JMLObjectSequence_rep"
"subsequence s1 i j \<equiv> JMLObjectSequence_cons3 (subsequence_isa (theSeq s1) i j) (j - i)"

trailer                :: "JMLObjectSequence_rep => JMLObjectSequence_rep"
" trailer s1 \<equiv> JMLObjectSequence_cons3 (trailer_isa (theSeq s1)) ((theLength s1) - 1)"

reverse                :: "JMLObjectSequence_rep => JMLObjectSequence_rep"
"reverse s \<equiv> JMLObjectSequence_cons3 (rev (theSeq s)) (theLength s)"

(* get the isa-representation of JMLObjectSequence *)
rep_to_isa :: "JMLObjectSequence_rep => JMLObjectSequence_isa"
"rep_to_isa s1 \<equiv> theSeq s1"
```

```
declare JMLObjectSequence_stdcons_def [simp]
declare JMLObjectSequence_cons2_def   [simp]
declare JMLObjectSequence_cons3_def   [simp]
declare concat_def                    [simp]
declare headerSeq_def                 [simp]
declare insertAfterIndex_def          [simp]
declare insertBack_def                [simp]
declare insertBeforeIndex_def         [simp]
declare insertFront_def               [simp]
declare prefix_def                    [simp]
declare removeItemAt_def              [simp]
declare removePrefix_def              [simp]
declare replaceItemAt_def             [simp]
declare singleton_def                 [simp]
declare subsequence_def               [simp]
declare trailer_def                   [simp]
declare rep_to_isa_def                [simp]

(* the set of all JMLObjectSequences *)
consts ObjSeqSet :: "JMLObjectSequence_rep set"

inductive ObjSeqSet
intros
std_constructor[intro!]: "JMLObjectSequence_stdcons \<in> ObjSeqSet"
constructor2   [intro!]: "JMLObjectSequence_cons2 e \<in> ObjSeqSet"
constructor3   [intro!]: "((s1 = []) = (length s1 = 0) \<and> (0 \<le> length s1)) ==>
                          JMLObjectSequence_cons3 s1 (int (length s1)) \<in> ObjSeqSet"

end
```

JMLObjectSequence_spec.thy - pre- and postconditions

```
left out, because similar to JMLObjectSequence in first approach
```

JMLObjectSequence_eqTheory.thy - equational theory

```
theory JMLObjectSequence_eqTheory = JMLObjectSequence_rep:

lemma eq_theory_1: "\<not> has (rep_to_isa JMLObjectSequence_stdcons) e1"
by simp

lemma eq_theory_2: "size (rep_to_isa JMLObjectSequence_stdcons) = 0"
by simp

lemma eq_theory_3: "size (rep_to_isa (JMLObjectSequence_cons2 e1)) = 1"
by simp

lemma eq_theory_4: "e1 \<noteq> Null ==> itemAt (rep_to_isa (JMLObjectSequence_cons2 e1)) 0 = e1"
by simp

lemma eq_theory_5: "e1 \<noteq> Null ==> first (rep_to_isa (insertFront s e1)) = e1"
by simp

lemma eq_theory_6:"e1 \<noteq> Null ==> size (rep_to_isa (insertFront s e1)) == size (rep_to_isa s) + 1"
by simp

lemma eq_theory_7: "e1 \<noteq> Null ==> last (rep_to_isa (insertBack s e1)) = e1"
by simp

lemma eq_theory_8: "e1 \<noteq> Null ==>
itemAt (rep_to_isa (insertBack s e1)) (int (size (rep_to_isa s))) = e1"
by simp

(* Taken from specification
// !FIXME! The following may be inconsistent: argument to itemAt might
// be equal to size, but it is required to be less than size.

lemma eq_theory_9: "-1 <= n & n < int (size s) & e1 \<noteq> Null  ==>
itemAt (insertAfterIndex s n e1) (n+1) = e1"
apply(simp)
apply(auto)
*)
```

```
lemma eq_theory_10: "-1 <= n & n < int (size (rep_to_isa s)) ==>
size (rep_to_isa (insertAfterIndex s n e1)) = (size (rep_to_isa s)) + 1"
apply(simp)
by arith

(*
lemma eq_theory_11: "0 <= n & n <= int (size (rep_to_isa s)) & e1 \<noteq> Null  ==>
itemAt (rep_to_isa (insertBeforeIndex s n e1)) n = e1"
apply(simp)
by arith
*)

lemma eq_theory_12: " 0 <= n & n <= int (size (rep_to_isa s)) ==>
size (rep_to_isa (insertBeforeIndex s n e1)) = (size (rep_to_isa s)) + 1"
apply(simp)
by arith

(*
lemma eq_theory_13: "(isPrefix s s2) = (\<forall> i. 0 <= i & i < int (length s) &
                                       ( (itemAt s i) \<noteq> Null & (itemAt s2 i) = (itemAt s i))
                                       |
                                       ( (itemAt s2 i) = Null & (itemAt s i) = Null)
                                       )"
apply(simp)
apply(auto)
*)

(*
lemma eq_theory_14: "(isSubsequence s s2) =
(length s <= length s2 & ( (isPrefix s s2) | (isSubsequence s (trailer s2))))"
apply(auto)
*)

(* the variable s is of type JMLObjectSequence_isa in this case *)
lemma eq_theory_15: "(isEmpty s) = ((size s) = 0)"
apply(simp)
apply(case_tac s)
by auto

(* the variable s is of type JMLObjectSequence_isa in this case *)
lemma eq_theory_16: "(isEmpty s) = (length s = 0)"
apply(simp)
apply(case_tac s)
by auto

lemma eq_theory_17: "equals (rep_to_isa (JMLObjectSequence_cons2 e1)) (
rep_to_isa (insertFront JMLObjectSequence_stdcons e1))"
by simp

lemma eq_theory_18: "equals (rep_to_isa (JMLObjectSequence_cons2 e1))
(rep_to_isa (insertBack JMLObjectSequence_stdcons e1))"
by simp

lemma eq_theory_19: "equals (rep_to_isa (JMLObjectSequence_cons2 e1))
(rep_to_isa (insertAfterIndex JMLObjectSequence_stdcons -1 e1))"
by simp

lemma eq_theory_20: "equals (rep_to_isa (JMLObjectSequence_cons2 e1))
(rep_to_isa (insertBeforeIndex JMLObjectSequence_stdcons 0 e1))"
by simp

(* helper lemma for eq_theory_21 *)
lemma isPrefix_eq [simp]:"isPrefix s s"
apply(induct_tac s)
apply(auto)
done

lemma eq_theory_21: "1 \<le> (size s) ==> equals s (insertFront_isa (trailer_isa s) (first s))"
apply(simp)
apply(case_tac s)
by auto
```

```
(*
lemma helper_22_1 [simp]:"isPrefix s (butlast s @ (last s)#[])"
apply(induct_tac s)
apply(auto)
done

lemma helper_22_2 [simp]:"1 \<le> length s ==> isPrefix ((last s)#[] @ (rev (butlast s))) (rev s)"
apply(simp)
apply(case_tac s)
apply(auto)


lemma eq_theory_22: "1 \<le> (size s) ==> equals s (insertBack (headerF s) (last s))"
apply(simp)
apply(auto)
by
*)

(*
lemma eq_theory_23: "(isProperSubsequence s s2) = ((isSubsequence s s2) & (\<not>(equals s s2)))"
apply(simp)
*)

lemma eq_theory_24: "isSupersequence s s2 = isSubsequence s2 s"
by simp

lemma eq_theory_25: "isProperSupersequence s s2 = isProperSubsequence s2 s"
by simp

end
```

JMLObjectSequence_inv.thy - class invariants

```
theory JMLObjectSequence_invariants = JMLObjectSequence_rep + Subtype:

(* JML: public invariant owner==null; *)
lemma owner_is_null: "s \<in> ObjSeqSet ==> owner s = Null"
apply(erule ObjSeqSet.induct)
apply(simp_all)
done

(* JML: public invariant length == length(); *)
lemma length_is_consistent: "s \<in> ObjSeqSet ==> theLength s = int (length (rep_to_isa s))"
apply(erule ObjSeqSet.induct)
apply(simp_all)
done

(* JML: public invariant theSeq != null ==> length == theSeq.length();*)
lemma length_is_consistent_not_null: "s \<in> ObjSeqSet ==> rep_to_isa s \<noteq> [] -->
theLength s = int (length (rep_to_isa s))"
apply(erule ObjSeqSet.induct)
apply(simp_all)
done

(* JML: public invariant length >= 0 *)
lemma length_gt_zero: "s \<in> ObjSeqSet ==> 0 \<le> theLength s"
apply(erule ObjSeqSet.induct)
apply(simp_all)
done

(* JML: public invariant theSeq == null <==> length==0; *)
lemma seq_length_equivalence: "s \<in> ObjSeqSet ==> ((rep_to_isa s)=[]) = (0 = theLength s)"
apply(erule ObjSeqSet.induct)
apply(simp_all)
done

(* helper lemma *)
lemma help_contains_null_elems [simp]: "s \<noteq> [] ==> \<not> null s"
apply(case_tac s)
apply(auto)
done

(* JML: isEmpty() ==> !containsNull *)
```

```
lemma contains_null_elems: "s \<in> ObjSeqSet ==> isEmpty (rep_to_isa s) --> \<not> containsNull s"
apply(erule ObjSeqSet.induct)
apply(simp_all)
done



(* JML: elementType <: type(Object)*)
(* NOT PROVED: BoolT is subtypeof ClassT Object and other have to be defined *)
(*lemma elemType_subtypeof_objType: "s \<in> ObjSeqSet ==> elementType s \<le> ClassT Object"
apply(erule ObjSeqSet.induct)
apply(simp_all)
apply(induct_tac e)
apply(auto)
apply(blast)
done
end
*)

(* JML: (\forall Object o; o != null && has(o); \typeof(o) <: elementType); *)
(* NOT PROVED: same arguments as in lemma above *)
(*lemma typeof_elems_le_ObjT:"s \<in> ObjSeqSet ==> ()"*)

(* JML: equational_theory invariant *)
(* here we had to put lemmas in from JMLObjectSequence_eqTheory *)

end
```