Ownership-based Program Verification in JIVE

Ronny Zakhejm
Master Thesis

Supervised by Prof. Peter Müller and Adam Darvas
Software Component Technology Group
ETH Zürich

19th March 2006

Abstract

The current version of JIVE uses a Hoare style logic based on a non-modular proof technique. Besides the limitation that the whole program must be known to the verifier, layered object structures are difficult to verify with a non-modular technique. The goal of my master thesis is to make a step towards modularity by changing the logic of JIVE to be based on ownership proof technique, which is a modular proof technique supporting layered object structures. Ownership proof technique's modularity is achieved by the use of universe type system annotations and by several restrictions imposed on the program. Since this proof technique relies on universe type system annotations the modified JIVE logic should be aware of universe type system specific properties. Full modularity requires further considerations, which are not addressed in this thesis.

Contents

1 Introduction

Surprisingly verification of software, although a research area for already 35 years, is still one of the big hot spots of research in computer science. Hoare in his famous paper [8] presented a logic for reasoning about programs using a weakest precondition calculus based on rules and axioms. As observed by Bornat in [14] (and Hoare himself hints at it in his paper), the Hoare axiom of assignment is only sound when distinct variable names refer to distinct storage locations. References introduce the problem of pointer aliasing, since pointers decouple variable names from storage locations and hence two distinct variable names could both refer to a single storage location. Many proposals have been made to solve the pointer aliasing problem, one of the first was made by Morris in [15]. In this thesis the store model of Poetzsch-Heffter and Müller's program logic [12, 13] is used.

Another reason for software verification still being a hot research topic is the evolution of specification. Hoare's target programming language was functional without the concept of stateful objects. He even assumed function calls to be side-effect free with respect to the caller function in order to receive sound reasoning. Thus it was sufficient to use only function pre- and postconditions as program specification. Object-oriented programming languages introducing the concept of stateful objects demanded new specification elements, the most important of which are object invariants specifying consistency constraints on the object state. Additionally the assumption of Hoare that meaningful programs can be written without having function calls to be side-effect full respective to caller methods is no more true in object-oriented languages, where side-effect full method calls changing the state of their receiver object are normally the main part of the program logic. This yielded in the additional need to specify what locations a certain method is allowed to alter.

The use of object invariants brought a new problem, since without invariants when proving a function to be correct, it was sufficient to consider only one proof obligation, namely that the precondition of the function in question is sufficient to derive the postcondition. As object invariants according to their semantics must hold in all visible states (a definition of visible state will be given in section 2.1.3), many more proof obligations arise, namely that the invariants of all objects being in a visible state must always hold. The problem with these new proof obligations is that the verifier can not determine the set of all objects being in a visible state as it would force the verifier to consider all possible program states that could occur at an invocation of the method in question, which is normally impossible. This problem could be circumvented if a differential approach is taken using the fact that at the beginning of a method the invariants of all objects in a visible state hold. The verifier could check that every assignment and invocation occuring in the method in question does not break the invariants of the objects in a visible state by examining all changed locations. Again pointer aliasing renders these checks very difficult, since a modification in an object can be visible at several (even unexpected) points.

Besides the problem, that it is normally very difficult to reason about the set of objects being in a visible state, it is even theoretically impossible to reason about this set for a library method, since when verifing that library code the client of this library and its objects are not known to the verifier.

The approaches to overcome these difficulties in verifying programs with invariants can be divided into two groups, namely the non-modular and the modular solutions. While the non-modular solutions target to verify whole programs, meaning that all parts of the code are known to the verifier, modular solutions aim at verifying well-defined program parts without the need to know the whole program. All solutions try to make a reasonable trade-off between introducing additional rules restricting the expressiveness of invariants and limiting the usage of certain language constructs on the one hand and the difficulty of proving the resulting proof obligations on the other hand.

The current version of the program verifier Jive [1, 2] uses a non-modular logic. Since this non-modular logic is severely limited in its ability of proving the correctness of programs having layered structures, it is the target of this thesis to describe a way how to make a big step toward Jive's logic being modular. This new logic is based on the ownership proof technique [3]. It uses the ownership model of the universe type system [6, 12, 19] which allows a modular logic to support layered structures.

In Section 2 the current version of Jive is presented by describing its architecture and its non-modular logic. Section 3 shows a major limitation of the current non-modular Jive logic and tries to get out the exact reason why this limitation emerges. Section 4 presents the concepts of the universe type system and its ownership model needed for introducing a new proof technique explained in Section 5 that should overcome the limitation shown in Section 3 by being a big step towards full modularity. Section 6 and section 7 deal with the integration of the concepts and techniques described in previous sections into Jive. While section 6 describes the formalization and axiomatization of the universe type system's ownership model in Jive and Isabelle, Section 7 is about how to integrate the new proof technique smoothly into Jive's existing framework. Section 8 tries to demonstrate the advantages of this thesis with examples. In section 9 some interesting details about the implementation of the previous sections is given and in section 10 future tasks still required for full modularity are listed.

2 Current Jive

2.1 Jive's Architecture

Jive stands for Java Interactive Verification Environment. It is, as its name says, an interactive program verifier based on a partial correctness Hoare-logic. Jive is being developed by the Software Component Technology Group at ETH Zürich together with the Softwaretechnik Group at TU Kaiserslautern and is implemented in Java.

While this thesis was in progress a concept called light ownership using the universe type system was independently of this thesis introduced into Jive's logic. Therefore when referring to current Jive's logic in this thesis the Jive logic without any universe type system support is meant.

2.1.1 Target Language of Jive

Jive's target language is Diet Java Card [17], which is a subset of Java Card [7]. Java Card itself supports most Java language constructs, but is sequential, i.e. there is only one execution thread. Although Diet Java Card supports less language constructs than Java Card, specially nesting of expressions is very restricted, it still supports all important object-oriented features like inheritance and dynamic binding of methods. It was introduced by the Jive developers in order to keep verification simple without lessening the power of Jive since Java Card programs can be transformed into Diet Java Card programs. As a sidemark it should be mentioned that the current logic of Jive does not support all of Diet Java Card language constructs but only a even smaller Java subset called Java-KEx [16]. Since most considerations performed in this thesis hold for both, Diet Java Card and Java-KEx input programs, the target language in this thesis will be assumed to be Diet Java Card. It will be explicitely mentioned if Java-KEx is assumed as target language for some considerations.

2.1.2 Java Modeling Language

Diet Java Card programs used as input for JIVE are to be specified with JML annotations. JML stands for Java Modeling Language [4, 5] and is used to specify programs. Its syntax is a subset of Java enriched with specification-specific expressions and statements. JML specification annotations can be written in the program code file itself or in a separate specification file. Figure 1 shows an example of a JML annotated Java program. There are many tools developed for JML of which some are included in the JML release. Among those included in the release is the JML checker, which performs the usual compiler checks on a JML annotated Java program in order to guarantee syntactical correctness and well-typed expressions. Additionally some more sophisticated semantic checks are done as for instance method purity checks[1] and side-effect freeness of specification expressions checks are performed. An other tool included in the JML release and building on the JML checker is the JML runtime assertion checker, which compiles the input JML annotated Java program to Java bytecode adding runtime checks for the executable parts of the specification.

The next section describes some basic JML elements used in this thesis. More information about JML can be found in [4, 5].

2.1.3 Some Basic JML Elements

In Figure 1 some basic JML elements can be seen.

Invariants specify the consistent states of objects. The term *invariant semantics* means the specification when invariants have to hold for what objects. The invariant semantics of JML requires the invariants of all objects to hold in all visible states. A precise definition of visible state can be found in [5, Section 8.2]. Simplified it can be said that a state is visible for object o if it occurs (a) at the end of a constructor of o, (b) at the beginning or at the end of a method invocation with o as receiver object or (c) when no constructor or method of o is in progress. Note that this definition of a visible state does not allow any method others that of o to break the invariant of o even temporarily. Since JIVE provides no mechanism to enforce conditions to hold after every statement, JIVE does not use the visible state definition of JML but a modified version:

[1]which are currently still unsound

Figure 1: Exampe of a JML annotated Java program

```
class JML_Example {
   /*@ public model String mydoublestring;
     @ public represents mydoublestring <- mystring + mystring;
     @*/

   private /*@ spec_public @*/ String mystring;
   //@ in mydoublestring;

   /*@ public invariant mystring != null;
     @ public invariant mystring.startsWith("something");
     @ public invariant mydoublestring != null; @*/

   /*@ public requires in != null;
     @ public ensures (\result.equals(\old(mystring) + in))
     @     && (mystring.equals(\old(mystring) + in));
     @ public assignable mydoublestring;
     @*/
   public String addToMyString(String in) {
      mystring = mystring + in;
      return mystring;
   }

   /*@ public requires true;
     @ public ensures \result == mystring;
     @ public assignable \nothing;
     @*/
   public /*@ pure @*/ String getMyString() {
      return mystring;
   }

   /*@ requires true;
     @ ensures \result.equals(in + in);
     @ signals (NullPointerException) in == null;
     @*/
   public /*@ pure @*/ String doubleString(String in) {
      return in + in;
   }

   public /*@ pure @*/ JML_Example() {
      mystring = new String();
   }
}
```

Definition 1 (Visible States) *A program execution* state *is called* visible *if it is a pre- or poststate of a method call. These states are also called* points of execution control transfer.

As this definition of a visible state does not take in account constructors, the invariant semantics in JIVE is changed to

Definition 2 (Invariant Semantics in current JIVE) *The invariants of all objects have to hold in all visible states, besides for the invariant of a newly created object at the beginning of its constructor.*

Note that every execution control transfer has two different points of consideration, namely in the caller and in the callee. While the definition of a visible state will remain unchanged, the invariant semantics will be further elaborated during this thesis.

The Invariant on line 9 in Figure 1 means that *mystring* is not allowed to be *null* in any visible state except for the prestate of its constructor.

Requires clauses, which are often called method precondition, specify the condition that must hold prior to an invocation of the specified method in order to receive a specified result[2]. The result of an invocation of the specified method if the precondition does not hold is unspecified and can be arbitrary. Note that even if the precondition is met exceptions can still be thrown.

The requires clause on line 31 in figure 1 states that method *doubleString* does not have any preconditions in order to execute in a specified way. Nevertheless, if *in* equals *null*, an exception is thrown.

Ensures clauses, often called method postcondition, specify what holds when the specified method terminated normally, i.e. not due to the throwing of an uncaught exception. The return value is expressed by the keyword $\backslash result$. An expression enclosed by the keyword $\backslash old$ refers to the value of the enclosed expression when evaluating the expression in the prestate of the method.

In figure 1 there are three ensures clauses, for every declared method one. The ensures clause on line 24 of method *getMyString* specify its return value to equal *mystring*.

[2]for a partial correctness logic only in the case the method terminates.

Signals clauses, also called exceptional postcondition, specify the condition when an uncaught exception of a specific type will be thrown. Additionally they specify the poststate of the method after its abrupt termination. Method *doubleString* in Figure 1 throws an exception of type *NullPointerException* if parameter *in* equals *null*. This behaviour is specified in the signals clause on line 33.

A method behavior specification is a set of requires, ensures and signals clauses. Several non-contradicting method behavior specifications can be given for a specific method. Those method behavior specifications are concatenated using the *also* keyword.

Assignable clauses, often referred to as modifies clauses, specify what locations the specified method may alter. All other locations must remain unchanged. A method mentioning the keyword *nothing* in its assignable clause is called a *pure* method and must not modify any location, i.e. the method is side-effect free. Alternatively a method can be specified directly to be pure by the keyword *pure*. Pure methods can create new objects and modify those objects freely. This purity semantics is often called weak purity, whereas strong purity does not permit any modifications on the heap. Note that although assignable clauses are also referred to as modifies clauses (and JML treats them equally), there is actually a minor semantic difference between them. While a location not mentioned in a modifies clause may be modified temporarily during method execution, a location not mentioned in an assignable clause must not be assigned during method execution even if the original value is restored. The condition what location may be assigned to is also called frame condition.

Model fields are fields only used for specification purposes and are declared with the modifier *model*. *Represents* clauses specify how values of model fields are evaluated using a mapping from concrete fields and other model fields to the specified model field. In Figure 1 such a represents clause can be found on line 3, where the model field *mydoublestring* is evaluated by concatenating the concrete field *mystring* with itself, *mystring*.

Every model field also declares a data group [9]. Data groups are sets of locations, to which fields, both concrete and model, can be added to using an *in* clause. A data group mentioned in an assignable clause means that the specified method can assign to every location which is a member of the mentioned data group.

In Figure 1 *mydoublestring* declares such a data group as it is a model field. The in clause on line 7 declares *mystring* to be a member of *mydoublestring*'s data group. In the assignable clause of method *addToMyString* only *mydoublestring* needs to be mentioned, although *mystring* is actually modified.

All specifications of a superclass are inherited by its subclasses. This is often called *behavioral subtyping*. Subclasses can add invariants and method behavior specifications to overridden methods, although these behavior specifications must not contradict those of the supertype's method specification (which are inherited). As assignable clauses guarantee that not mentioned locations are not modified, assignable clauses are not permitted in a specification of an overriding method. To allow locations introduced in a subclass to be modified in the overriding method, these subclass locations must be added to data groups of the supertype mentioned in the assignable clause of the supertype's method.

For further description of JML see [4, 5]. Data groups are explained in [9].

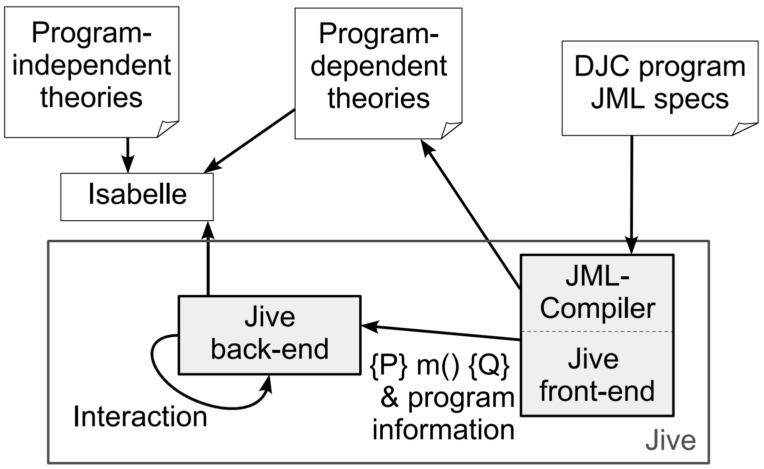
2.1.4 Jive Front-End

Figure 2 illustrates the architecture of Jive. Jive is split into a front-end and a back-end. The front-end is the first part of Jive and the back-end the second. The names are somewhat missleading as the back-end is the interactive part whereas the frond-end runs without any user interaction. Currently the front-end and the back-end are separate programs due to incompatibilities between different versions of Java.

Jive takes a Diet Java Card program as input annotated with JML specifications. In the front-end the JML compiler is called for parsing and typechecking the program. The front-end takes the generated syntax tree and produces the following output:

- Proof obligations for each method occuring in the program. Those proof obligations are in the form of Hoare triples, i.e. {Prestate condition} method() {Poststate condition}.
- Program information for the back-end, including code structure and type information.
- Program-dependent theories containing program information as for instance the occuring types, their subtype relationship and declared fields. These theories are later used by the theorem prover.

Figure 2: Jive's Architecture


Program-independent theories
Program-dependent theories
DJC program JML specs
Isabelle
JML-Compiler
Jive front-end
Jive back-end
{P} m() {Q} & program information
Interaction
Jive

2.1.5 Jive Back-End

The Jive back-end is the place where actual verification happens. In the back-end, which is interactive, Jive's rules and axioms transform the Hoare triples received from the front-end into first-order logic implications, which can be exported to a theorem prover. The back-end is interactive since the user can specify which rules or axioms should be applied and help Jive to prove the program, although there are predefined automatic proof tactics implemented in Jive, as for example a practical weakest precondition tactic.

2.1.6 Isabelle

Those former mentioned first-order logic implications are exported to the theorem prover Isabelle[10]. Additionally to those implications Isabelle takes as input the program-independent theories which contain the Isabelle formalization of the used store model and the program-dependent theories generated by the Jive front-end containing program specific information. Now either those implications exported to Isabelle are successfully proven or they could not be proven and the user must provide additional information in the back-end as for example stronger loop invariants or apply different rules and axioms in the verification calculus in order to do a retry of the Isabelle proof.

It is planned for the near future to use the non-interactive theorem prover Simplify[11] as an additional theorem prover. First the implications are given to Simplify and only those implications Simplify does not succeed to prove are forwarded to Isabelle.

2.2 Current Jive logic

In order to verify Diet Java Card programs with the help of a theorem prover several things are needed:

- A formalization of several aspects of the target language including its object store model and its arithmetics.
- A mapping from JML specification into first-order logic formulas using the chosen object store model formalization.
- A set of Hoare rules and axioms allowing a transformation from Hoare triples into first-order logic implications using the chosen object store model formalization described in the first point.

Every point will be described in the following sections omitting those parts not relevant to this thesis.

2.3 Jive's Object Store Model

To formalize properties of the object store the store model of Poetzsch-Heffter and Müller's program logic is used. It is formalized in multi-sorted first order logic with recursive datatypes. Here only those parts relevant to this thesis are presented. For a further description of the object store model of Poetzsch-Heffter and Müller's program logic and its axiomatization see [12, 13].

A store model comprises sort Value, Type, Location and Store.

Sort *Value* represents (a) values of a primitive type as integer or boolean, (b) references to objects, either 'regular' objects or array objects or (c) the `null` reference denoted by $nullV$. Diet Java Card values map directly to sort Value.

Sort *Type* contains (a) primitive types as integers or boolean, (b) class types, either regular types or array types and (c) the type of the `null` reference. Diet Java Card types map directly to sort Type. The reflexive, transitive subtype relation is denoted by $\preceq$. The function $typeof : Value \rightarrow Type$ returns the type of a Value. Note that as $typeof$ is reflexive, when the term "S is a subtype of T" is used in this thesis, it means that S is either equal T or T is a supertype of S.

Object states are modeled with *Locations* representing instance variables. An object has a Location for each field of its class. There is a sort *FieldId* for unique field identifiers of a program. Two functions are needed: (a) a function *loc* yielding the corresponding Location of a given Value and a given FieldId (or $undefined$ if this Location does not exist) and (b) a function *obj* returning the corresponding Value of a given Location. These functions are declared as follows:

$$\begin{array}{rcl} loc & : & Value \times FieldId \rightarrow Location \cup \{undefined\} \\ obj & : & Location \rightarrow Value \end{array}$$

$loc(X,\ f)$ is abbreviated $X.f$.

Sort *Store* models object stores supporting operations to read and to update locations, to create new objects and to test whether an object is allocated.

$$\begin{array}{rcl} \_\langle\_ := \_\rangle & : & Store \times Location \times Value \rightarrow Store \\ \_(\_) & : & Store \times Location \rightarrow Value \\ new & : & Store \times ClassId \rightarrow Value \\ \_\langle\_\rangle & : & Store \times ClassId \rightarrow Store \\ alive & : & Value \times Store \rightarrow bool \end{array}$$

$OS\langle X := Y\rangle$ returns the Store resulting from storing Value Y at Location X in Store OS. The read operation is expressed by $OS(X)$ reading Location X in Store OS and returning the read Value. For new object allocations there are two functions. While $new(OS, certainId)$ yields the reference to the newly created object of the type determined by $certainId$, $OS\langle certainId\rangle$ yields the new Store after an object creation of type $certainId$. Finally function $alive(Y, OS)$ returns true if and only if Y is allocated in OS.

The constant symbol \$ of sort Store is used in formulas to refer to the current object store. The prestore of a method execution is the object store immediately after the arguments of the method are evaluated, but before the computation of the precondition. The poststore of a method execution is defined in an analogous way namely it is the object store after the execution of the return statement, but before the evaluation of the postcondition.

These sorts and their axiomatization are defined in the program-independent Isabelle theories.

2.4 JML Transformation

As explained in Section 2.2 a mapping from JML specification into first-order logic formulas using the previously explained object store model is needed in order to receive the prestate and the poststate condition of the verification calculus. Note that prestate and poststate condition do not mean the specified pre- and postcondition (requires and ensures clauses) of the method in question, but a much more general thing. Most JML annotation elements contribute either to the prestate condition, to the poststate condition or even to both. This section describes how JIVE maps JML annotation elements into conjuncts of the prestate or the poststate condition.

2.4.1 Notation

Some notation first. The γ function denotes the function transforming a JML expression into object store model logic formulas of JIVE. The method itself is not described here since it is not relevant for this thesis. For a comprehensive description see [21, 2]. $formula[X/Y]$ means $formula$ with every free occurence of Y substituted by X, e.g. $(f(Y) \ \wedge \ Y \ \vee \ \forall \, Y \ . \ g(Y))[X/Y]$ means $f(X) \ \wedge \ X \ \vee \ \forall \, Y \ . \ g(Y)$. Y in $\forall \, Y \ . \ g(Y)$ is not substituted since Y is not free. \$ is a constant of sort store and means the current store, as already defined earlier.

A special program variable χ is used in the poststate to indicate whether the method in question terminated normally ($\chi = normal$) or abruptly by throwing an uncaught exception ($\chi = exc$). In the first case, the predefined program variable $resV$ holds the resulting value of the (non-void) method. In the case of an abrupt method termination the predefined program variable $excV$ holds a reference to the exception.

While requires clauses contribute to the prestate condition, the transformed ensures and signal clauses are conjoined to the poststate condition. The transformation of requires, ensures and signals clauses are not needed for this thesis and hence are not described here. For a description of their transformation see [2].

2.4.2 Invariants

As pointed out in the introduction, the invariants of all object must not be broken in a visible state and hence must be considered while verifiying a specific method. For naming reasons every class is assigned a unique number from 1 to n and Ti stands for the name of the i-th class. The translation of the invariants is done as follows for every class. The invariants of class say C are transformed using the γ function. These transformed invariants are then conjoined to one formula called I_C for class C. A function $InvC$ of type $Value \rightarrow Store \rightarrow bool$ is defined as follows:

$$InvC(X, OS) \;\equiv\; typeof(X) \preceq C \;\Longrightarrow\; I_C[X/this, OS/\$]$$

$InvC(X, OS)$ means that if X's type is a subtype of C then the invariants of class C must hold for Value X. Note the substitutions in I_C, where $this$ is substituted by X since the invariants are formulized for $this$ but used here for X. Additionally \$ is substituted by OS as the invariants are formulized for the current store \$ but used here for the specified store parameter OS. This transformation is done for every class Ti occuring in the input program resulting in n $InvTi$ functions.

What is left is quantifying over all objects the invariants of which have to hold according to the invariant semantics. Since the invariant semantics of current Jive mentiones constructors, an introduction into Diet Java Card object initialization is needed here. The only constructor allowed in Diet Java Card is the default constructor. Non-default constructors are simulated by the default constructor and a special constructor-like $init()$ method to be invoked just after the default constructor call. The fact that at the beginning of the constructor-like $init()$ method call the newly created object's invariant does not hold[3] is

[3]unless the default values of its state satisfy its invariant.

reflected in current JIVE's invariant semantics. For notational reason a *normal* method is called a method not being a constructor-like $init()$ method.

From the invariant semantics it can be followed that

1. At an execution control transfer not being the beginning of a constructor-like $init()$ method call the invariants of all objects have to hold.
2. At an execution control transfer being the beginning of a constructor-like $init()$ method call the invariants of all objects besides the newly created object have to hold.

Hence two different quantifications must be used, one that quantifies over all objects and one that quantifies over all objects but one specific.

The function handling most cases by quantifying over all objects is $INV(OS)$ of type $Store \rightarrow bool$ which is defined as follows

$$\begin{aligned} INV(OS) \;\equiv\; & \forall\, X :: Value\ .\ (alive(X, OS)\ \wedge\ X \neq nullV \implies \\ & InvTi(X, OS)\ \wedge\ ...\ \wedge\ InvTn(X, OS)) \end{aligned}$$

INV means that for a certain Store all Values which are alive and not null must satisfy all guarded invariants. Note that this way invariant inheritance is guaranteed.

This approach is not modular since a quantification over all objects requires the knowledge of the whole program while verifying.

At the pre- and poststate of a normal method no constructor-like $init()$ method is involved. Hence INV is conjoined to both, the prestate and the poststate condition of a normal method as the invariants of all objects must hold at both places.

Let's look at an example on how invariants are transformed. In figure 3 the two invariants on line 3 & 4 are first transformed with the γ function to

$$\gamma(this.a < this.b) \text{ and } \gamma(this.a + this.b > 10)$$

Figure 3: Translation example

```
class Translation_example {
   public int a, b;
   /*@ invariant a < b;
     @ invariant a + b > 10;
     @*/
}
```

Then they are conjoined and this conjuntion is called $I_{TransEx}$.

$$I_{TransEx} \equiv \gamma(this.a < this.b) \wedge \gamma(this.a + this.b > 10)$$

$InvTransEx(X, OS)$ is then defined as

$$\begin{aligned} InvTransEx(X, OS) &\equiv typeof(X) \preceq Translation\_example \implies \\ &\quad I_{TransEx}[X/this, OS/\$] \\ InvTransEx(X, OS) &\equiv typeof(X) \preceq Translation\_example \implies \\ &\quad \gamma(X.a < X.b) \wedge \gamma(X.a + X.b > 10)[OS/\$] \end{aligned}$$

The function quantifying over all but one objects is called $INVC(OS, X)$ (C for constructor), is of type $Store \rightarrow Value \rightarrow bool$ and is defined as

$$\begin{aligned} INVC(OS, X) &\equiv \forall\, Y\ .\ (X \neq Y \wedge alive(Y, OS) \wedge Y \neq nullV \\ &\quad \implies InvTi(Y, OS) \wedge ... \wedge InvTn(Y, OS)) \end{aligned}$$

In the prestate of constructor-like $init()$ methods the invariants of all objects but the new created hold. Hence $INVC$ is conjoined to the prestate condition of a constructor-like $init()$ method with $this$ as value parameter. In the poststate of a constructor-like $init()$ method the invariants of all objects have to hold. Hence INV is conjoined to its poststate condition.

Figure 4 shows the invariant semantics in JIVE that before and after an invocation statement ($o.p()$) $INV(\$)$ must hold, besides at the prestate and poststate of the method in question itself ($c.m()$). The example of figure 4 will be used through this thesis having a method $m()$ of type C being the method to be

Figure 4: Invariant semantics in current JIVE, shown for classes O and C

```
class O {
    {assume INVC($,this)}
    void init() { }
    {assert INV($)}

    {assume INV($)}
    void p() { }
    {assert INV($)}
}

class C {
    O o;

    {assume INV($)}
    void m() {
        ...
        o = new O();
        {assert INVC($,o)}
        o.init();
        {assume INV($)}
        ...
        {assert INV($)}
        o.p();
        {assume INV($)}
        ...
    }
    {assert INV($)}
}

class Main {
    void main() {
        C c = new C();
        c.m();
    }
}
```

verified. This method is called as $c.m()$, where c is sometimes referred to as *this* since it is the receiver object of the method to be verified. Additionally a method invocation $p()$ on receiver object o of type O occures in $c.m()$.

The functions defined in this section as $InvTi$, INV and $INVC$ are part of the program-dependent theories.

2.4.3 Assignable clauses

First the concept of a downward closure is explained in short. The downward closure of a field is the set of all locations this field depends on. In the case of a concrete field this is trivially one location, but a model field can depend on several locations. The downward closure of a set of field is the union of the downward closures of the elements of this set. For more details see [2].

Since JIVE provides no way of stating conditions to hold after every statement, assignable clauses are handled the modifies way.

Assignable clauses contribute to both, the prestate and the poststate condition. While in the prestate condition the current store \$ is saved into a logical variable S and the downward clausure of the set of all locations mentioned in assignable clauses is stored into a logical variable M, a check is added to the poststate condition that a location was either not alive in S, mentioned in M or not modified. For details see [2].

Note that this transformation is not modular, since the computation of the downward closure requires that all subclasses of the class to be verified are already known while verifying.

2.4.4 Hoare Triples

Since in this thesis normally all clauses but invariants do not have to be considered separately, it is sufficient to consider their conjunction. So the prestate condition will be

$$\{INV(\$) \ \wedge \ P\}$$

for a normal method and

$$\{INVC(\$, this) \ \wedge \ P\}$$

for a constructor-like $init()$ method, where P is the conjunction of all contributions to the prestate condition but invariants. Analogously the poststate condition is

$$\{INV(\$)\ \wedge\ Q\}$$

for all methods, where Q is the conjunction of everything but the invariants. So a Hoare-triple of a method looks like

$$\{INV(\$)\ \wedge\ P\}\ T : m(par)\ \{INV(\$)\ \wedge\ Q\}$$

for a normal method and

$$\{INVC(\$, this)\ \wedge\ P\}\ T : init(par)\ \{INV(\$)\ \wedge\ Q\}$$

for a constructor-like $init()$ method.

2.5 Selected Rules and Axioms

Section 2.2 explained that a set of Hoare rules and axioms allowing a transformation from Hoare triples into first-order logic implications using the chosen object store model formalization is needed. Here only selected rules and axioms are described, since only those are relevant to this thesis. [16, section 4 and especially section 4.3] is a comprehensive source of JIVE's Hoare-logic in general and the rules and axioms particularly.

2.5.1 Axiom of Cast

The axiom of cast is defined as

$$\left\{\begin{array}{l} (typeof(e) \preceq T\ \wedge\ \mathrm{P}[e/x]) \vee \\ (typeof(e) \not\preceq T\ \wedge\ \mathrm{P}[\$\langle\mathrm{CastExc}\rangle/\$, new(\$, \mathrm{CastExc})/excV]) \end{array}\right\}\ x = (T)e;\ \{\mathrm{P}\}$$

A cast either passes or fails. In the case the cast passes it is known that in the prestate of the cast e's type should be a subtype of T and that all properties of x that should hold in the poststate of the cast should hold in the prestate for e. In the case the cast fails e's type is not a subtype of T in the prestate of the cast and a new instance of CastExc is assigned to $excV$, the logical variable containing the thrown exception.

This axiom will later on be referred to as cast-axiom.

2.5.2 Rule of (non-void) Invocation

The rule of non-void invocation on a non-null target is

$$\frac{\{Pre\} \quad C : m(par) \ \{Post\}}{\{c \neq nullV \ \wedge \ Pre[c/this, e/par]\} \ x = c.m(e) \ \{Post[x/resV]\}} \tag{1}$$

meaning that if a method $m(param)$ exists for type C of which c's type is a subtype with specified prestate and poststate conditions $\{Pre\}$ and $\{Post\}$ respectively, then it is known that if in the prestate of the method invocation c does not equal *null* and the specified prestate condition for the invoked method $C : m(param)$, with *this* substituted by the target c and with the specified parameters *par* substituted by the actual parameters e, holds in the prestate of the method invocation, then the specified poststate condition of the invocated method $C :: m(param)$, with $resV$ substituted by x, holds in the poststate of the method invocation.

Note that in case $C : m(param)$ is a normal method $INV(\$)$ is part of both the specified prestate and poststate condition meaning that the invariants of all objects (including the *this* object) must hold before and after the method invocation! In case m is a constructor-like $init()$ method, $INVC(\$, this)$ is part of the prestate condition. As *this* is substituted by the target c in the prestate condition, which in case of a $init()$ method is the newly created object, $INVC$'s meaning is that the invariants of all objects but the newly created have to hold. The invariant part of the poststate condition for a $init()$ method is $INV(\$)$.

This rule will later on be referred to as invoc-rule.

2.5.3 Rule of Void Invocation

The rule of void invocation on a non-null target is very similar to the rule of non-void invocation except that there is no return value and hence no return value substitution.

$$\frac{\{Pre\} \quad C : m(par)\ \{Post\}}{\{c \neq nullV \ \wedge \ Pre[c/this, e/par]\}\ c.m(e)\ \{Post\}} \tag{2}$$

This rule will later on be referred to as invoc-void-rule.

2.5.4 Axiom of New Object Creation

The axiom of new object creation is

$$\{\mathrm{P}[new(\$, T)/x, \$\langle T\rangle/\$]\}\ x = \mathrm{new\ T}();\ \{\mathrm{P}\}$$

since the assignment of a newly created object requires the prestate of the assignment to contain the poststate condition of the assignment with two substitutions. In the first one the assigned variable x is substituted by the newly created object denoted by $new(\$, T)$ and in the second one the current store \$ is substituted by $\$\langle T\rangle$, which is the store resulting from the addition of the newly created object to the current store.

This rule will later on be referred to as new-axiom.

3 Limitation of JIVE's Non-Modular Logic

3.1 The Problem

As sketched in the introduction, the non-modular logic of JIVE is not able to handle layered object structures. The invariant semantics requires the invariants of all objects to hold at all points of execution control transfer if no constructor-like $init()$ method is involved.

Let's consider the case illustrated in figure 5 , where a class $CounterWithCache$

Figure 5: Limitation of Jive's non-modular logic

```
class Counter {
    private /*@ spec_public @*/ int c;

    //@ ensures this.c == \old(this.c) + 1;
    //@ assignable c;
    {assume INV($)}
    void inc() {
        c = c + 1;
    }
    {assert INV($)} // INV must hold but violated !!
}

class CounterWithCache {
    private /*@ rep @*/ Counter counter;
    int cache;

    //@ invariant counter != null;
    //@ invariant cache == counter.c;

    {assume INV($)}
    void addOne() {
        {assert INV($)}
        counter.inc();
        {assume INV($)} // INV is assumed but violated !!
        cache = cache + 1;
    }
    {assert INV($)}
}
```

has got a field *counter* of type *Counter*. The invariant of *CounterWithCache* depends on the state of the *Counter* instance captured in its *counter* field. Additionally it is assumed that *CounterWithCache* is not leaking, i.e no reference to *counter* is given to the outside of *CounterWithCache*. Furthermore *Counter* does not have any direct or indirect reference to *CounterWithCache*. Note that this example is not in Diet Java Card but in Java for brevity reasons. All considerations are also valid for the equivalent Diet Java Card example.

When considering *CounterWithCache*'s method *addOne* with receiver object say *counterWithCache* it will be noticed that the *counter.inc*() call temporarily violates *counterWithCache*'s invariant which is re-established in the next line. The code is nevertheless sensible as *counter*'s *inc* will execute safely although it violates *counterWithCache*'s invariant since *counter* can not notice *counterWithCache*'s invariant violation as it has no (transitive) reference to *counterWithCache*. But Jive's logic fails in proving this example, since after the call of *counter.inc*() on line 14 the invariants of all objects, including *counterWithCache*, are assumed to hold, but *counterWithCache*'s invariant is violated. To solve that problem without changing the invariant semantics there are two possibilities, which are both not feasible.

1. *counter* should have a reference to *counterWithCache* and increment *counterWithCache*'s *cache*. But *Counter* instances could be used in different contexts with different invariant constellations. It is completely unfeasible that *Counter* should take in account all peculiarities of its clients, even those already existing[4].

2. *counterWithCache* should not use *inc*() to increment *counter* but assign directly to *counter*'s *c* field. This solution violates information hiding since *c* is private and should be private as it is internal representation and hence *counterWithCache* may not assign to it.

The core of the problem is that it is a layered object structure. When *Counter* is written, it is not aware of its clients. On the other hand clients of *Counter* use *Counter* instances as part of their state and therefore their invariants should mention *Counter*'s instances' state as well. These clients modify instances of *Counter* using method calls rather than direct field updates for reason of information hiding. The modifications in the state of *Counter*'s instance due to those method calls may violate (temporarily) the state of *Counter*'s clients. Hence methods of *Counter* must be allowed to violate its clients invariants as *Counter* does even not know its clients.

[4]If considering clients not yet existing a non-modular approach can not work since the whole program must be known in advance.

3.2 A First Approach

Before trying to solve that problem, a definition is introduced.

Definition 3 *The method a statement of which is currently in progress is called the* currently executing method*. Its receiver object is called the* currently executing object*. A method being a transitive caller of the currently executing method is called an* executing method*, whereas its receiver object is called an* executing object[5]*. Analogously a method being neither the currently executing method nor a transitive caller of the currently executing method is called a* non-executing method*. An object is called a* non-executing object *if non of its methods are executing.*

A possible approach allowing *Counter*'s methods to violate the invariants of its clients is to change the invariant semantics in the following manner. The new proof obligation would be that the invariants of all non-executing objects must hold upon a execution control transfer (besides some $init()$ method exceptions). This solution solves the problem of figure 5 since $counterWithCache$ is executing when $counter.inc()$ is in progress. The problem with this solution are re-entrant method calls reaching an object in an inconsistent state. This happens in the case when a transitive callee of an executing method $o.m()$ calls a method of o, say $o.foo()$. o's invariants might be broken upon the $o.foo()$ call as o is already an executing object.

There are two solutions: either re-entrant method calls are forbidden or upon a re-entrant method call the invariant of this method's receiver object must be re-established although it is an executing object. Both solutions require an exact knowledge of all executing methods forcing the verifier to consider all invocation sites of the method in question, i.e. this solution is, besides being very complicated, not modular. Using a conservative assumption that every object might be executing when considering the need to re-establish the invariant of the receiver object at a method call site leads back to JIVE's logic.

The approach used in this thesis is to separate the object structure of the heap into different layers unaware of their layers above. That is exactly what the universe type system does. The universe type system will be used to layer the object structure enabling a modification of the invariant semantics in order to get much weaker but yet sound proof obligations.

[5]With an intuitive understanding of a stack the currently executing method is the method on top of the stack and an executing method is a method executing somewhere on the stack.

4 Universe Type System

This section is a short summary of [6, Section 3] and [3, Section 5] omitting parts not relevant to this thesis. For a full description of the universe type system including a formalization and a type safety proof see [12, 19].

4.1 Ownership Model

Each object is directly owned by at most one other object called its *owner*. A *context* is the set of all object directly owned by the same object. The set of objects having no owner is called the *root context*. The owner-relationship is acyclic and since every object has got at most one owner the contexts form a tree with the root context as root. The owner object is specified at object creation and can not be changed during the lifetime of that object. Context Ω is a child of context Ψ if the owner of the objects in Ω is a member of context Ψ.

Object A is the owner of context Ω if A is the owner of all objects in Ω. An object X is inside context Φ if X is an element of Φ or of one of Φ's descendant contexts. Otherwise X is outside context Φ. If B is the owner of D then B owns object D. An object E is inside the context of F if E is inside the context F is an element of. An object F is in the same context as C if both have the same owner. E is inside the context owned by A if A is its transitive owner. See figure 6 for an illustration. In Appendix A many more ownership relationship notations are listed.

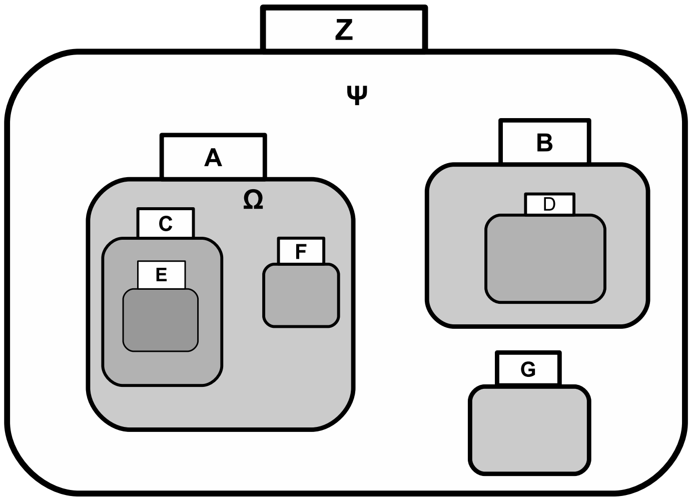
Contexts are used to restrict references between objects, namely objects are only allowed to have write references to objects in the same context or to directly owned objects.

4.2 Ownership Types and Subtyping

There are three different reference types in the universe type system:

- `peer` references are references between objects in the same context. This reference is the default if no universe type is specified.
- `rep` references are references from an object to a directly owned object.

Figure 6: Ownership contexts



- `readonly` references are references between arbitrary objects.

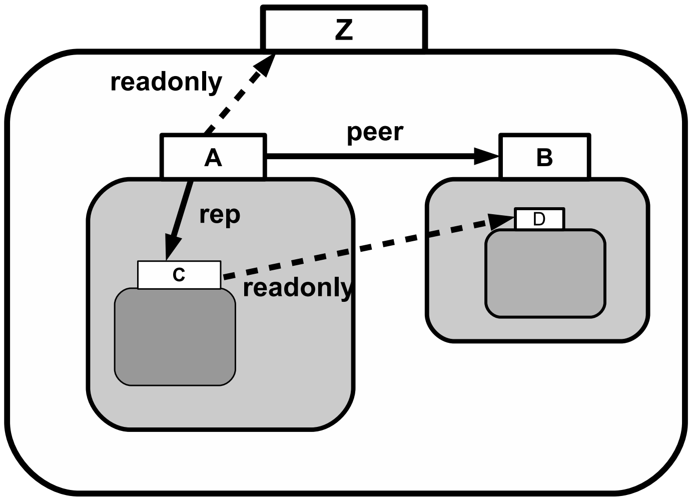
While peer and rep references are write references, readonly references may only be used for reading. See figure 7 for illustated examples.

Universe types are part of the type system. Hence the type of every reference, i.e. fields, locals, parameters and return types consists of a universe type and a Diet Java Card type, e.g. `rep Object`, `readonly List` or `peer String`. This way every expression (if not of a primitive type) has got a static type consisting of the universe type and the Diet Java Card type. Primitive types do not have an ownership modifier since they are not references. Arrays have two ownership modifiers, one for the array object and one for its elements. All arrays of a multi-dimensional array belong to the same context. Further array considerations are not needed for this thesis. The reader is referred to the former mentioned references.

The subtype relationship of the universe types is as follows. `rep` and `peer` are subtypes of `readonly`. `rep` and `peer` are not in any subtype relationship between themselves.

A type T is a subtype of type S if the universe type of T is a subtype of S's universe type and T's Diet Java Card type is a subtype of the Diet Java Card

Figure 7: Universe references



type of S.

Examples:

- `rep Object` is a subtype of `rep Object`
- `peer String` is a subtype of `peer Object`
- `rep List` is a subtype of `readonly List`
- `peer Collection` is a subtype of `readonly Object`
- `readonly String` is *not* a subtype of `peer Object`
- `rep Object` is *not* a subtype of `readonly Socket`

Figure 8 illustrates by an example of a double-liked list the usage of the universe type system. Class *Node* has got `peer` references to *next* and *prev* since all nodes are in the same context. This context is owned by an instance of Class *List*, thus *first* and *last* are of universe type `rep`. The objects to store in the list can be in arbitrary contexts and thus the *data* field of *Node* has universe type `readonly`.

Figure 8: Universe type example

```
class Node {
    /*@ peer @*/ Node next, prev;
    /*@ readonly @*/ Object data;
}

class List {
    /*@ rep @*/ Node first, last;

    void add(/*@ readonly @*/ Object o) {
        /*@ rep @*/ Node toAdd = new /*@ rep @*/ Node();
        last.next = toAdd;
        last = toAdd;
    }
}
```

4.3 Type Rules and Dynamic Information

Assignments are handled the standard way, meaning that the type of the right-hand side must be a subtype of the type of the left-hand side. Example: a `peer String` reference can be assigned to a `readonly String` left-hand side.

Object creation requires to specify the type of the object to create, i.e. both the Diet Java Card type and the universe type. Example: `x = new rep T(..)`. Readonly objects can not be created since every object must be assigned to a context.

Down-casts as `x = (rep String) roString` with `roString` of type `readonly String` require a check whether the object to cast is of the appropriate universe type. This is done by storing the owner of an object upon creation in a special field called *owner*. This so-called *owner*-field is declared in the supertype of all objects, namely in *java.lang.Object*. A rep cast means that the object to cast is either *null* or the *this* object is the owner of the object to cast: `roString == null || roString.owner == this`. Analogously, a peer cast means that the object to cast is either *null* or the owner of the *this* object is the owner of the object to cast: `roString == null || roString.owner == this.owner`.

Field accesses. The universe type of a field access (as `y = x.f`) is determined considering both, x and f.

1. If only peer references are involved, everything happens in the same contexts and thus $x.f$ is of type `peer`.
2. If the universe type of f is `rep` and x equals $this$, then the owner of $this.f$ is $this$ and hence the type of $x.f \equiv this.f$ is `rep`.
3. If x is of type `rep` and f is of type `peer`, then the type of $x.f$ is `rep` since the owner of x is $this$ and the owner of f is the same as the owner of x.
4. All other cases can not be evaluated at compile type and hence $x.f$ is of universe type `readonly`.

The universe type of field updates are computed analogously. Field updates on readonly objects are not allowed.

It follows from these four points that an field access chain with one field access of type `readonly` renders the universe type of the whole expression to `readonly`.

Method call's return type has got a universe type part and therefore universe type consideration of method calls is analogous to field accesses with x the receiver object and f the specified return type. Method calls on readonly references must be pure methods and taking only readonly parameters. For further considerations of method calls, specially on universe types of parameters see the references mentioned above.

4.4 Universe Invariant

The universe type system guarantees the *universe invariant.*

Definition 4 (Universe Invariant) *For every execution state if object X holds a direct reference to object Y then at least one of the following cases applies:*

1. *X and Y are in the same context.*
2. *X is the owner of Y.*
3. *The reference is readonly.*

With respect to this *universe invariant*, local variables and formal parameters behave like instance variables of the *this* object.

Note that the universe invariant ensures the following properties (*universe invariant corollaries*):

Corollary 1 (Universe Invariant Corollaries)

1. *Every write reference chain*[6] *from an object outside Ω to an object inside Ω passes through Ω's owner.*
2. *There is no write reference chain from inside a context Ω to an object outside Ω.*
3. *A method of an object member of context Ω can only be directly invoked by another member of Ω or by Ω's owner.*
4. *A method executing in context Ω, i.e. the this object is a member of context Ω, can only modify locations of objects inside Ω, either directly by field update or indirectly by method invocations.*

Hence an object c owning the context Ψ has got full control over its owned objects, as no object outside Ψ can modify an object inside Ψ without calling (directly or indirectly) a method of c.

5 Ownership Proof Technique

5.1 Review of Example

Let's take again a look at figure 5. As explained in section 3, *Counter* cannot know anything about its clients. As these clients use *Counter* instances as internal representation and hence mention fields of these *Counter* instances in their invariants, modifications of the state of those instances of *Counter* break the invariants of these clients. The invariant semantics must allow methods of *Counter* to break the invariants of its clients.

In section 3.2 an approach to weaken the invariant semantics was presented, which allows a method to break the invariant of an executing object. It was

[6] A reference chain is a concatenation of several references. For example x.f.g.h is a reference chain, from object x through reference to object x.f, then through reference g to object x.f.g and finally through reference h to object x.f.g.h.

also pointed out that the big problem with this approach are re-entrant method calls. As the receiver of a re-entrant method call can have a broken invariant, the verifier is required to know what objects are executing in order to know whether the receiver's invariant must be re-established prior to the re-entrant method call. This requirement, besides being very difficult, renders this method non-modular.

The desired solution should not consider at verification time what objects have currently got broken invariants in order to guarantee that the receiver object of a method call does not have a broken invariant[7]. Its because such solutions are neither easy nor modular. Furthermore clients should be allowed to integrate the state of their representation objects into their invariants[8] and hence representation object's methods should be allowed to break the invariants of their clients.

The universe type system opens up a perspective to solve this problem in a modular way. The universe invariant guarantees that no write reference chain exists from inside a context Ω to an object outside Ω. In order to guarantee that no method outside a context Ω is called by a method inside Ω, a first restriction is introduced.

Definition 5 (Relevance) *An object y inside the context of an object x is also called an object* relevant to x.

Restriction 1 (Relevance Restriction) *Readonly method calls occuring in a method of object c element of context Ω must be on a receiver object inside Ω, i.e. the receiver object must be relevant to c.*

The universe invariant together with this restriction guarantee that to method of an object outside Ω is called from a method inside Ω. Hence the owner of Ω (which is outside Ω) can assume that a method invocation with receiver object o of universe type rep (meaning that o is owned by the owner of Ω and hence an element of Ω) cannot result in a re-entrant method invocation to an object outside Ω and hence not to Ω's owner.

[7]Surely this consideration must be done when developing the solution. But the resulting proof technique should be free of considerations about what objects have currently got broken invariants.

[8]For an easy solution not allowing objects to integrate the state of their representation objects into their invariants see [3, Section 3].

Since the universe invariant guarantees as well that every non-readonly method invocation chain[9] from an object outside Ω to an object inside Ω passes through Ω's owner, a method of Ω's owner knows that no object inside Ω is an executing object since otherwise the method would be a re-entrant method call from an object inside Ω to an object outside Ω.

Summarized the following properties follow from the universe invariant, together with restricted readonly calls:

Corollary 2 (Restricted Readonly Calls Corollaries)

1. *There are no method calls from inside a context Ω to an object outside Ω, hence no re-entrant calls upon a rep call or a readonly call with a receiver not element of Ω can occur.*
2. *All transitive owners of an object X are executing objects when a method of X is called as non-readonly call*[10].
3. *A rep object is not an executing object. That is actually a special case of the more general statement, that no object inside Ω but not element of Ω is an executing object.*

The two first points allows a method to break the invariants of its transitive owners without any danger. The third point states that only method calls to an object in the same context can be itself a re-entrant call or can result in an re-entrant call to the caller. Modular verification gets closer...

5.2 New Invariant Semantics

Some expressions often mentioned in this sections should be named.

Definition 6 (Constructing object) *The receiver object of a constructor-like init() method is called the* constructing object.

[9]A method invocation chain here does not mean a.foo().bar().add(), but rather that method foo() calls method bar() and method bar() calls method add().

[10]The case were a non-readonly method was called from a readonly method can actually occur when new object were created in this readonly method. Although then their might be non-executing transitive owners, these transitive owners can not depend on these new objects as the heap modifications of the readonly method are not visible outside the readonly method. Hence for the further consideration one can assume that point 2 holds.

Definition 7 (Begin of Init) *The execution state at the beginning of a call to a constructor-like $init()$ method is called the* begin of init.

The invariant semantics of the solution of section 3.2 was that the invariants of all non-executing objects must hold upon a execution control transfer, besides the invariant of the constructing object at its begin of init. With help of the universe type system this proof obligation can get weaker, since the universe invariant can give guarantees on certain objects whether they are executing or whether re-entrant calls can occur.

The universe invariant guarantees that for a method of object x with x element of context Ω all methods of objects outside Ω cannot be called. Objects inside Ω but not element of Ω are non-executing and hence their methods can be called freely since their invariants are ensured by the invariant semantics. Only method calls on receiver objects being an element of Ω are a potential danger for re-entrant method calls.

A solution to prevent re-entrant method calls on receiver objects with broken invariants is to require all objects in the same context as the *this* object to have an intact invariant even if they are executing. With this new requirement no re-entrant method calls can occur on a receiver object with a broken invariant. Additionally this solution requires a method $m()$ to re-establish at most those invariants that were intact at the beginning of $m()$ (besides its own invariant in case $m()$ equals $init()$). Hence a new invariant semantics must be formulated.

Definition 8 (New Invariant Semantics)

1. *The invariants of all non-executing objects hold upon an execution control transfer.*
2. *Additionally to 1 the invariants of all objects which are element of the current execution context must hold upon an execution control transfer unless this invariant belongs to the caller of a non-peer*[11] *method invocation.*
3. *The only exception to rule 1 and 2 is that the invariant of the constructing object at its begin of init can have broken invariants, although it is not executing and additionally it might be an element of the current execution context.*

Point 2 is explained by the fact that the caller will be executing during the called method. Additionally point 1 of the restricted readonly calls corollaries

[11] A non-peer method invocation means that the receiver of the method call is not an element of the caller's context. Rep call are surely non-peer calls. Readonly call are non-peer calls if the receiver is actually not an element of the caller's context.

guarantee that the caller will not be an element of any execution context while the called method or one of its (transitive) callees are in progress.

This new invariant semantics will from now on be referred to as the invariant semantics.

Note that if the invariant semantics is taken directly as proof obligation it is not modular since the quantification is done over all non-executing objects.

5.3 Restrictions and Admissibility

The next step is to see what parts of the proof obligation can be automatically ensured by the universe invariant or additional restrictions. Specially the non-modular parts of the proof obligation must be already ensured in order to get modular proof obligations.

Some notation. The method in question is called $m()$ while its *this* object is called c and is an element of Ω. Its owned context is called Ψ.

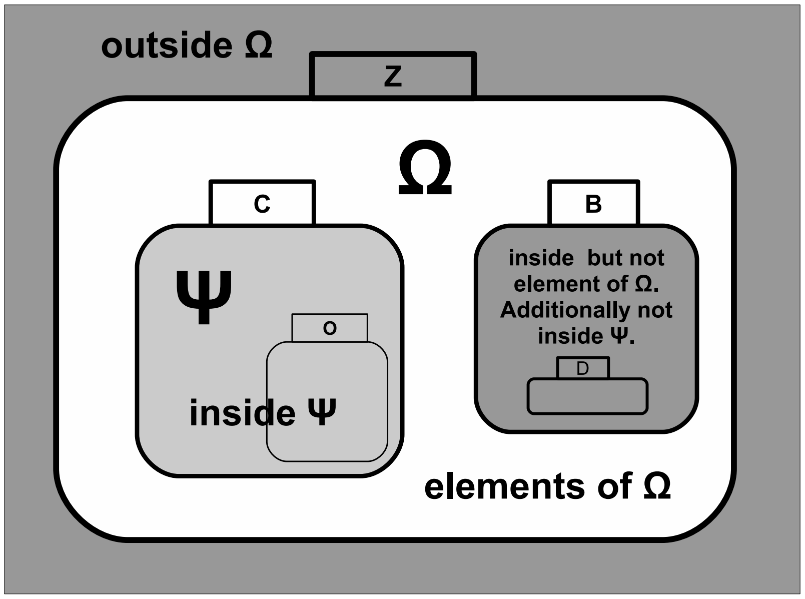
Since an arbitrary non-executing object and not known while verifying $m()$ can include the state of c in its invariant, any changes to the state of c can break the invariant of this arbitrary object. Since this arbitrary object is not known while verifying $m()$, this arbitrary object's invariant will be broken although its is non-executing. Hence invariants must be restricted in such a way that all objects not known to a modular verifier are either executing or their invariants hold.

Let's consider what objects may mention c in their invariants.

1. The universe invariant guarantees that all transitive owners of c are executing and therefore their invariants are allowed depend on c.
2. All other objects outside Ω are either surely[12] or possibly[13] non-executing, hence it must be ensured that their invariants do not depend on c.
3. Furthermore invariants specify the consistent state of an object and therefore invariants should not depend on objects outside their context. Hence no object inside Ψ is allowed to depend on c.

[12] Objects which are not in the same context as a transitive owner of c.
[13] Objects which are in the same context as a transitive owner of c.

Figure 9: Different contexts to consider for admissibility



If the consideration is done the other way round, i.e. on what objects may the invariant of c depend, it can be said that the invariant of c should depend neither on objects outside Ω (point 3) nor on objects inside but not element of Ω (point 2) except for objects in Ψ (point 1). Or defined in a positive way:

> The invariant of object c element of context Ω may depend only on members of Ω or on objects inside the context owned by c.

Consider figure 9 for an illustration. This condition, on what objects an invariant may depend, is called *admissibility* condition. An invariant is *admissible* if it satifies the admissibility condition. Since invariants are formed out of expressions, the term admissible is used, with the same meaning as for invariants, for expressions as well.

The above mentioned admissibility condition is actually used in [3, Section 9] for the visibility technique (with additional subclass separation and visibility restrictions for soundness and modularity). In this thesis the used admissibility condition is even more restrictive, namely

Restriction 2 (Invariant Peer Restriction) *The invariant of object c element of Ω is not permitted to depend on members of Ω except for c itself.*

This further restriction is sensible as invariants specifying the consistent state of an object should only depend on locations controlled by the invariant's object. Now a first version of the *ownership admissibility condition* can be formulated:

Definition 9 (Ownership Admissibility Semantics) *The invariant of object c element of context Ω may depend only on fields of c or on fields of objects inside the context owned by c.*

As the final version of the ownership admissibility condition will be defined more technically, this first version is called the ownership admissibility semantics as it illustrates what locations invariants may depend on.

5.4 Subclass Separation

Since invariants may depend on the state of objects inside their owned context, subclassing could lead to a potential unsoundness with rep objects accessible from both, the superclass and the subclass. Consider figure 10 . Again this example is in Java for brevity reasons.

Class *Super* and specially method *bad*() are verified easily since *Super* has got no invariant[14]. But if *bad*() is called on a *Sub* object *sub* and *sub.subT* equals *sub.superT*, *sub*'s invariants is broken. Figure 11 illustrates the situation when *sub.subT* equals *sub.superT*.

To avoid this unsoundness, there are two possibilities. Either all methods of all superclasses have to be verified again when verifying the subclass, which violates modularity, or the set of objects reachable from rep fields of the superclass must be disjoint from the set of objects reachable from rep fields of the subclass. This is achieved by taking the following measures.

- All rep fields must be declared private. (Renders line 28 in figure 10 illegal since *superT* must be declared private)
- Every method taking a rep parameter or returning a rep value must be declared private. This makes it impossible for *Super* to call *Sub*'s *setSubT* with *superT* as parameter.

[14]See the next section for the proof technique.

Figure 10: Subclass separation example

```
class T {
    private /*@ spec_public peer @*/ Object out;
    //@ modifies out;
    public void setOut(/*@ peer @*/ Object o) {
        out = o;
    }
}

class Super {
    protected /*@ rep @*/ T superT;

    public /*@ readonly @*/ T leak() {
        return superT;
    }

    public void bad() {
        superT.setOut(null);
    }
}

class Sub extends Super {
    private /*@ spec_public rep @*/ T subT;

    /*@ public invariant subT != null
      @               && subT.out != null @*/

    public void separate() {
        subT = superT;
        subT = (rep T) leak();
    }

    public void setSubT(/*@ rep @*/ T t) {
        subT = t;
    }
}
```

Figure 11: Subclassing problem

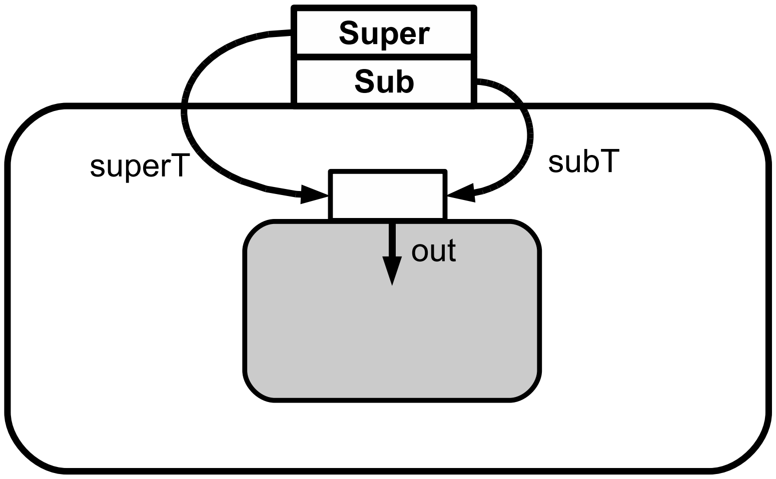
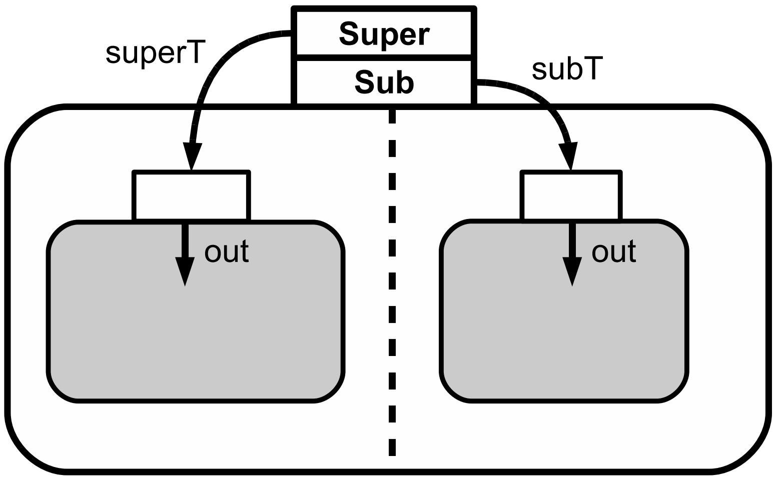


Figure 12: Subclass separation



- The cast on line 29 should fail although *leak*() returns an object having *this* as owner. Therefore additional information is needed, namely the declared class must be stored at object creation in a special field *declClass* of the created object. The field *declClass* is declared in the supertype of all types, *java.lang.Object*. Since for every object creation statement it is statically known in which class it occurs, it is no problem to store this type object in the newly created object. The condition for the cast to pass is `roRef == null || (roRef.owner == this && roRef.declClass == actClass)`, where `roRef` denotes a readonly reference and `actClass` the class the cast appears in.

- In case of a peer cast it must be assured that the object to cast is declared in the same class as the *this* object. Hence the cast condition is `roRef == null || (roRef.owner == this.owner && roRef.declClass == this.declClass)`.

Taking these measures *Super* cannot modify any objects reachable from rep fields of *Sub* and vice versa. See figure 12.

Since *Sub* does not have control over the fields of *Super*, these fields should not be used in *Sub*'s invariants. Hence all invariant expressions of *Sub* having more that one field access must start with a field declared in *Sub*.

5.5 Ownership Admissibility

On page 39 a first version of the ownership admissibility condition was formulated. In order to receive the final version, the following remarks should be considered.

- Constant fields cannot change. Hence if constant field accesses are added to an admissible expression of c's invariant, the resulting expression is still under full control of c and thus admissible.

- Static fields and static invariants are currently not covered by the universe type system and hence invariants are not allowed to be static or to depend on static fields unless this static field is a constant.

- Since it is out of scope of this thesis to reason about locations a method invocation depends on, method calls are not permitted in invariants.

- Currently all fields are only concrete fields. Model fields are considered later.

The final version of the *ownership admissibility condition* can now be formulated [3].

Definition 10 (Ownership Admissibility Condition) *An invariant of class C is* ownership admissible *if all of its non-constant expressions are of one of the following forms:*

1. *A non-static field access g_o with field g_0 declared in class C.*
2. *A non-static field access chain $g_o. \dots .g_N$ with $N > 0$, where g_0 is a rep field declared in class C and $g_1, \dots, g_{N-1}$ are fields declared as rep or peer.*
3. *A field access chain $g_o. \dots .g_N$ with $N > 0$, where an i exists so that $0 \leq i < N$, $g_o. \dots .g_i$ is of form 1 or 2 and $g_{i+1}, \dots, g_N$ are constant fields.*

Array accesses are treated as field accesses.

5.6 Ownership Proof Technique

5.6.1 Guaranteed Proof Obligations

This section tries to elaborate what parts of the proof obligations emerging from the invariant semantics are already guaranteed by the universe invariant together with admissible object invariants.

Again an object c element of context Ω executing a method $m()$ is considered. Additionally a method invocation $p()$ on receiver object o occurs in method $m()$ (see figure 9). The context owned by c is again called Ψ.

All objects are grouped into five distinct sets of objects (consider figure 6):

1. Objects outside Ω being a transitive owner of c.
2. Objects outside Ω and not a transitive owner of c.
3. Objects element of Ω. Note that c belongs to this group.
4. Objects inside Ψ.

5. Objects inside Ω, but neither element of Ω nor inside Ψ.

To show that the invariants of a certain group are not broken by $m()$, it must be shown that the invariants of this group

- hold at the beginning of $m()$ (denoted by Begin).
- are not broken by a field update in $m()$ (denoted by Field update).
- still hold after a method invocation $o.p()$ in $m()$ (denoted by Invoc).

Note that Init and Invoc are points of execution control transfer.

Newly created objects do not have to be considered until they are created. After their initialization their invariants hold and therefore the considerations for Field update and Invoc performed for objects already existing from the beginning of $m()$ are also valid for newly created objects after their initializations. Diet Java Card requires that the constructor-like $init()$ method is called immediately after its receiver's creation. But at that point the invariant of the receiver is allowed to be broken according to point 3 of the invariant semantics. Hence newly created object need not be handled separately in the following considerations.

Recall the universe invariant guaratees that a field update can only occur on objects in the same context as *this* or on objects directly owned by *this*. Let's consider which invariant can be broken by $m()$ but should hold at an execution control transfer in $m()$.

Group 1 According to point 1 of the restricted readonly calls corollaries all transitive owners of c are executing (and they are not element of Ω). Hence their invariants can be freely broken by $m()$ and need not be further considered.

Group 2 An object *out* in group 2 is either executing or non-executing. *out* is not an element of Ω, therefore in case its is executing its invariant can be freely broken by $m()$ and need not be further considered. For the further consideration it is assumed that *out* is non-executing.

- Begin: As *out* is non-executing, its invariant hold at the beginning of $m()$ because of point 1 of the invariant semantics.

- Field update: As *out* is not a transitive owner of c, admissibility of invariants guarantees that *out*'s invariant does not depend on locations inside Ω. Since a field of an object outside Ω can not be modified by an execution inside Ω (point 4 of universe invariant corollaries), *out*'s invariants are not broken by a field update in $m()$.
- Invoc: As *out* is still non-executing after $o.p()$, its invariant hold after $o.p()$.

Group 3 Because of point 2 of the invariant semantics the invariant of an object *peer* in group 3 must hold at a point of execution control transfer although *peer* might be executing. Only if *peer* equals c and the point of execution control transfer is a rep call, *peer*'s invariant need not hold.

- Begin: As *peer* is element of Ω, its invariant hold at the beginning of $m()$ due to point 2 of the invariant semantics, unless $m()$ is $init()$ and *peer* equals c.
- Field update: A field update in $m()$ can break the invariant of *peer*.
- Invoc: As *peer* is element of Ω, its invariant hold after $o.p()$, unless *peer* equals c and o is of universe type rep.

Group 4 According to point 3 of the restricted readonly calls corollaries an object *rep* in group 4 is non-executing and therefore its invariant must hold at a point of execution control transfer in $m()$.

- Begin: As *rep* is non-executing, its invariant hold at the beginning of $m()$.
- Field update: A field update in $m()$ can break the invariant of *rep* if *rep* is directly owned by c.
- Invoc: As *rep* is still non-executing after $o.p()$, its invariant hold after $o.p()$.

Group 5 According to point 3 of the restricted readonly calls corollaries an object *peerrep* in group 5 is non-executing.

- Begin: As *peerrep* is non-executing, its invariant hold at the beginning of $m()$.

- Field update: Admissibility of invariants guarantees that the invariant of *peerrep* depends neither on locations element of Ω nor on locations inside Ψ. Hence a field update in c can not break the invariant of *peerrep*.
- Invoc: As *peerrep* is still non-executing after $o.p()$, its invariant hold after $o.p()$.

The remaining proof obligations are that at a point of execution control transfer it must be assured that

> the invariant of all objects element of Ω hold. Only if this point of execution control transfer is a non-rep call, the invariant of c need not hold.
>
> the invariant of all objects element of Ψ hold.

If field updates are restricted to target objects element of Ω, the second proof obligation can be omitted. This restriction is actually imposed on $m()$ in the visibility technique in [3, Section 9], where the first of these remaining proof obligations is actually the proof obligation (including some visibility considerations to make the proof obligations modular). For this thesis a even stronger restriction is imposed on $m()$:

Restriction 3 (Field Updates Target) *All field updates occuring in method $m()$ must be on target this.*

Let's consider again field updates for group 3 & 4.

- Group 3: As a field update can only occur on c, the invariants of all other elements of Ω can not be broken by a field update in $m()$ as their invariants may not depend on c.
- Group 4: Field updates on receiver object from group 4 are not allowed.

Hence the only invariant that can be broken by a field update or a method call in $m()$ is the invariant of c itself. Hence the proof obligations are modular !

The following very important observation follows from this fact that $m()$ can only break c's invariant.

Observation 1 (INVC always holds) *At any execution point of $c.m()$ executing in context Ω no invariants of any object inside Ω can have a broken invariant except for c itself or a newly created object the $init()$ method of which was not yet called. In other words: Besides a newly created object the invariants of all objects relevant to c hold at any point of execution during $c.m()$, besides the invariant of c itself.*

Corollary 3 *If at an execution point in $c.m()$ there is no newly created object the $init()$ method of which was not yet called, then if the invariant of c holds, the invariants of all objects relevant to c hold.*

5.6.2 Elaboration of The Ownership Proof Technique

So the ownership proof technique can be elaborated by looking at c's invariant at different points of execution control transfer in $c.m()$:

1. At the beginning of $c.m() \neq c.init()$ c's invariant is guaranteed by point 2 of the invariant semantics.
2. At the beginning of $c.m() = c.init()$ point 3 of the invariant semantics allows c's invariant to be broken
3. At the beginning of a method invocation $o.p()$ inside $c.m()$ point 2 of the invariant semantics states that the invariant of c only has hold if o is an element of Ω. Otherwise the invariant of c is allowed to be broken. Additionally it must be guaranteed that o is not outside Ω, i.e. o is relevant to c (restriction 1 in section 5.1).
4. After a method invocation $o.p()$ inside $c.m()$, point 2 of the invariant semantics states that the invariant of c holds if o is an element of Ω. Otherwise c's invariant could be broken.
5. At the end of $c.m()$ point 2 of the invariant semantics requires c's invariant to hold. But since c's invariant could be broken, it must be re-established.

Note that a peer and a rep reference are always relevant.

A notation: $INV_{Rel}(x)$ denotes that the invariants of all objects relevant to x hold. $INVC_{Rel}(x)$ denotes that the invariants of all objects relevant to x hold except for the invariants of x. Note that $INVC_{Rel}(c)$ always holds during execution of $c.m()$ and that therefore if the invariant of c hold then $INV_{Rel}(c)$ holds.

Now all information is gathered to formulate the *ownership proof technique* [3]:

1. At the beginning of $c.m()$ it can be assumed that $INV_{Rel}(c)$ holds.
2. At the beginning of $c.init()$ it can be assumed that $INVC_{Rel}(c)$ holds.
3. At the beginning of a method invocation $o.p()$ with o of universe type peer the invariant of c must be re-established.
4. At the beginning of a method invocation $o.p()$ with o of universe type rep nothing must be assured.
5. At the beginning of a method invocation $o.p()$ with o of universe type readonly the invariant of c must be re-established if o is in the same context as c. Additionally it must be checked that o is relevant to c.
6. After a method invocation $o.p()$ it can be assumed that $INV_{Rel}(o)$ holds, meaning that if o is peer c's invariants hold.
7. At the end of $c.m()$ the invariant of c must be re-established.

Figure 13 shows an example of the proof obligations emerging of the ownership proof technique.

5.7 Model Fields

Model fields depend on other fields. So if model fields are metioned in invariants, all fields the model fields depends on must be considered while checking that invariant's admissibility. This approach is not modular as model fields are normally declared in (abstract) superclasses and their mapping is only defined by a represents clause in subclasses not known to the verifier. Therefore a different solution is needed, namely that model field must be restricted to depend only on locations the access expressions of which are surely admissible in all invariants that may mention the model field.

Let's consider again the ownership admissibility semantics: Invariants of object c element of context Ω may depend only on fields of c or on fields of objects

Figure 13: Ownership Proof Technique Example

```
class C {
    rep O repO;
    peer O peerO;
    readonly O readonlyO;
    {assume INV_Rel this} // Prestate (1)
    void m() {
        ...
        {assure INV of this} // Peer call (3)
        peerO.p();
        {assume INV_Rel peerO} // Call Return (6)
        ...
        {assure true} // Rep call (4)
        repO.p();
        {assume INV_Rel repO} // Call Return (6)
        ...
        {assure readonlyO is relevant to this && // Readonly call (5)
         if readonlyO is actually peer then INV of this}
        readonlyO.p();
        {assume INV_Rel readonlyO} // Call Return (6)
        ...
    }
    {INV of this} // Poststate (7)
}
```

inside the context owned by c. So the model field could either be a model field of c or a model field an object inside the context owned by c. In the first case the model field is allowed to depend on the same locations as the invariant is allowed to. In the later case the model field could actually depend as well on these locations. But since the model field can be used in an invariant of the class O declaring the model field or in one of its subclasses, it may only depend on locations the invariant of O may depend. Hence represents clauses should be checked with the same admissibility checks as invariants are.

As a side-mark it should be mentioned that it is possible to determine while checking a represents clause in class C of a rep model field f whether the class M f is declared in mentions f in its invariants. M is known to the verifier because C must be a subtype of M. If f is not mentioned in M's invariants it is known that f could only be mentioned in an invariant of a transitive owner of C's instances, since subclasses of M are not allowed to mention f in their invariants because of subclass separation. Hence a different admissibility condition could be used for represents clauses, namely that rep model fields of object c element of context Ω and not mentioned in any invariant of c may depend only on fields of objects inside Ω.

6 Ownership in Jive and Isabelle

6.1 Formalizing Ownership

The universe type system is based on its ownership model. The information of the ownership structure is stored at two places, namely in the universe types and in the (hidden) owner field. The universe types enforce the universe invariant statically, i.e. without any need for runtime checks to guarantee universe type safety.

On the other hand the information stored in the owner field is indispensable for checking casts at runtime or to prove them statically. Furthermore, the universe invariant could be guarateed without typechecking the universe types but only with the owner field together with invariants and assertions checked at runtime. See figure 14 for an example and [6, Section 2] for a description how this would be done exactly.

The target of this thesis is to make Jive a big step nearer to modularity by integrating the ownership proof technique into Jive. Since the ownership proof

Figure 14: Second approach

```
class Node {
    /*@ peer @*/ Node next, prev;
    /*@ readonly @*/ Object data;

    /*@ invariant next == null || next.owner == this.owner;
      @ invariant prev == null || prev.owner == this.owner;
      @*/

    //@ requires n == null || n.owner == this.owner;
    void setNext(/*@ peer @*/ Node n) {
        next = n;
    }

    //@ ensures \result == null || \result.owner == this.owner;
    /*@ peer @*/ Node getNext() {
        return next;
    }
}

class List {
    /*@ rep @*/ Node first, last;

    /*@ invariant first == null || first.owner == this;
      @ invariant last == null || last.owner == this;
      @*/

    void add(/*@ readonly @*/ Object o) {
        /*@ rep @*/ Node toAdd = new /*@ rep @*/ Node();
        //@ assert toAdd == null || toAdd.owner == this;
        last.next = toAdd;
        //@ assert toAdd == null || toAdd.owner == this;
        last = toAdd;
        //@ assert toAdd == null || toAdd.owner == this;
    }
}
```

technique is based on the universe type system, it must be found a way how the information the universe type system provides about the ownership structure is formalized in Jive and how this formalization can be used to axiomatize the ownership model.

The two places where universe information is stored can inspire this formalization. So either the universe type is integrated into Jive's type system or the formalization is done with the owner field. The axiomatization of the universe invariant must be done according to the chosen formalization. Note that universe type safety itself is guaranteed, independently of the chosen formalization, by the JML compiler used in the front-end of Jive.

Let's discuss some advantages and disadvantages of these two formalizations. The approach integrating universe types into Jive's type system benefits from the formalization of the Jive type system as for example the knowledge of every locations type at any point in the program. Hence the universe type of every location is known at any point in the program without any proof effort. On the other hand no detailed ownership information is available, since with universe types only peer and rep relationships can be expressed. Additionally no ownership is available for readonly variable, although this information could be tracked from previous statements due to substitutions in Hoare rules. Without ownership information about readonly variables no universe cast can be proven. Furthermore this approach misses universe information specified in the program itself in invariants and pre- and postcondition.

The second approach uses the owner field together with additional specifications in invariants, pre- and postconditions to provide ownership information about locations. This means that only at points of execution control transfer ownership information is provided for free. At all other points in the program ownership information must be derived from points of execution control transfer by using the Hoare-style rules of Jive. Additionally the invariants, pre- and postconditions providing the ownership information can not just be used for free but must be proven as well. The derivation of ownership information from points of execution control transfer and the proof of ownership-specific invariants, pre- and postcondition are in most cases unnecessary as the derived information is already known or the truth of the specification is already guaranteed by the universe type system.

As these two approaches are both not suitable, let's state requirements on the desired formalization.

- It should not loose any ownership information available from all parts of the source program including universe types and ownership information in specifications.

- It should have proof obligations which are as weak as possible, meaning that the verifier should not be forced to prove things again that are already assured by, for example, typechecking.
- It should reduce modifications to the existing rules and formalisms of JIVE to a minimum, specially conceptual changes should occur as little as possible.

The approach taken in this thesis tries to consider all of these points.

Including universe types into JIVE's type system would cause big changes to the existing formalism as the universe type system actually uses multiple inheritance. For example `peer String` is a subtype of both, `peer Object` and `readonly String`, although these two are not in a subtype relationship. And introducing multiple inheritance into the formalization of the object store model is beyond the scope of this thesis.

The approach taken in this thesis is to introduce an *owner* function in Isabelle of type $Value \rightarrow Value$ returning the owner of the Value parameter. This solution is similar to the approach based on the *owner* field[15] but tries to avoid the usage of additional invariants, pre- and postconditions.

Let's consider how this *owner* function is used to store all available ownership information without the aid of unnecessary proof obligations. Lets define two helper functions, *peer* and *rep*. *rep* is of type $Type \rightarrow Store \rightarrow Value \rightarrow Value \rightarrow bool$, whereas the type of *peer* is $Store \rightarrow Value \rightarrow Value \rightarrow bool$. Again the class where an object o was created in is stored in o's field $declClass$.

$$\begin{aligned} peer\ T\ OS\ X\ Y &\equiv X = nullV \ \vee\ (owner\ X = owner\ Y\ \wedge\ OS(X.declClass) = T) \\ rep\ OS\ X\ Y &\equiv X = nullV \ \vee\ (owner\ X = Y\ \wedge\ OS(X.declClass) = OS(Y.declClass)) \end{aligned}$$

$peer\ T\ OS\ X\ Y$ is true iff X is null or its owner is Y's owner and X was created in class T. $rep\ OS\ X\ Y$ is true iff X is null or its owner is Y and X was created in the same class as Y. The store parameter is used for the field accesses $X.declClass$ resp. $Y.declClass$.

[15]and indeed the *owner* function could be defined as just being a short hand to the *owner* field.

In section 4.1 it was described that object in the root context have got no owner. In order to be able to reason about objects in the root context using the *owner* function, it will be defined that the special Value *nullV* is the owner of all objects in the root context. Hence every value (besides *nullV*) has got a well-defined owner.

As opposed to Diet Java Card arrays are not supported in Java-KEx. The formalization of ownership is done without considering arrays and hence for the rest of this section the target language is assumed to be either Java-KEx or Diet Java Card without arrays.

Fields Instead of using invariants to store the universe type of the field, Isabelle axioms are used. Hence for a rep field f and a peer field g of class C the following lines are emitted as axioms in the program-dependent theories.

$$\begin{array}{lcl} \mathit{typeof}\ X \preceq C\ \wedge\ \mathit{alive}\ \mathit{OS}\ X\ \wedge\ X \neq \mathit{nullV} & \Longrightarrow & \mathit{rep}\ C\ \mathit{OS}\ \mathit{OS}(X.f)\ X \\ \mathit{typeof}\ X \preceq C\ \wedge\ \mathit{alive}\ \mathit{OS}\ X\ \wedge\ X \neq \mathit{nullV} & \Longrightarrow & \mathit{peer}\ \mathit{OS}\ \mathit{OS}(X.g)\ X \end{array}$$

This axioms states that for every value X being a subtype of C, being alive and not null its f attribute is owned by X respectively its g attribute is owned by the owner of X. These lines could be merged into one line.

$$\begin{array}{l} \mathit{typeof}\ X \preceq C\ \wedge\ \mathit{alive}\ \mathit{OS}\ X\ \wedge\ X \neq \mathit{nullV} \\ \Longrightarrow \mathit{rep}\ C\ \mathit{OS}\ \mathit{OS}(X.f)\ X\ \wedge\ \mathit{peer}\ \mathit{OS}\ \mathit{OS}(X.g)\ X \end{array}$$

Note that shadowing is not a problem since in Jive unique attribute IDs are used to access f and g. For a readonly field no ownership information is available and thus nothing is emitted.

Locals and Parameters Since locals and parameters do not occur in any program-dependent theory, the method used for fields can not be applied to locals and paramters. Instead, ownership information of locals and parameters of the method in question is added to the implications exported to Isabelle. Hence the ownership information provided by the universe type of a local or a parameter is available (at least on the Isabelle level) without any proof effort.

Return values Normally the ownership information delivered by the universe type of a method's return value is preserved in the assigned variable's[16] universe type, but there are situations where this is not the case. So let's find these cases.

- If the assigned variable is of universe type rep or peer, then the universe type of the return value is preserved.
- If the declared universe type of the return value is readonly, no ownership information is available for the return value's universe type and hence nothing can be done.

So if the assigned variable is of universe type readonly and the declared universe type of the return value is peer or rep, there is ownership information available which is not preserved in the universe type of the assigned variable. The available information in case of a statement `readonly x = c.m()` is:

- If the declared return value is of universe type peer, it is known that $peer\ \$\ x\ c$ holds.
- If the declared return value is of universe type rep, it is known that $rep\ st.typeof(this)\ \$\ x\ c$ holds.

$\$$ denotes the store at in the poststate of the return value assignment, $st.typeof(this)$ means the static type of $this$, i.e. the class the method call occurs. Note that this information is available even if c's universe type is readonly. Hence the invoc-rule must be changed to add this information when needed.

So the poststate condition of the invoc-rule (rule (1)), which was

$$\{Post[x/resV]\}$$

should now be when the universe type of x is readonly

$$\{Post[x/resV]\ \wedge\ rep\ st.typeof(this)\ \$\ x\ c\}$$

in case of a rep return value and

$$\{Post[x/resV]\ \wedge\ peer\ \$\ x\ c\}$$

in case of a peer return value.

[16]In Diet Java Card the assigned variable must even be a local.

Universe up-casts In case of a universe up-cast ownership information is lost, since in a universe up-cast the assigned variable is always of universe type readonly. But as the equality of the right-hand side and the left-hand side of the cast can be preserved, the lost universe type information can be reconstructed from that equality. Hence nothing must be saved upon a universe up-cast.

Universe down-casts A universe down-cast passes only if the appropriate universe type constraint holds respectively fails if the constraint does not hold. Consider the cast `x = (rep T) y` appearing in class C, where y's universe type is readonly. The constraint for this cast is *rep* C \$ x c, meaning that y is either null or its owner is *this* and y was created in a method of class C (subclass separation). Analogously, if it were a peer cast, the constraint would be *peer* \$ x c. Hence the cast-axiom is changed to

$$\left\{ \begin{array}{l} (Cast_{pass} \ \wedge \ \mathrm{P}[e/x]) \vee \\ (Cast_{fail} \ \wedge \ \mathrm{P}[\$\langle \mathrm{CastExc}\rangle/\$, new(\$, \mathrm{CastExc})/exc]) \end{array} \right\} \ x = (T)e; \ \{\mathrm{P}\}$$

where $Cast_{pass} = \neg Cast_{fail}$. For a rep cast $Cast_{pass}$ and $Cast_{fail}$ are

$$\begin{array}{rcl} Cast_{pass} & \equiv & typeof(e) \preceq T \ \wedge \ rep\ st.typeof(this)\ \$\ x\ c \\ Cast_{fail} & \equiv & typeof(e) \not\preceq T \ \vee \ \neg rep\ st.typeof(this)\ \$\ x\ c \end{array}$$

where $st.typeof(this)$ denotes the static type of $this$, and for a peer cast $Cast_{pass}$ and $Cast_{fail}$ are defined to be

$$\begin{array}{rcl} Cast_{pass} & \equiv & typeof(e) \preceq T \ \wedge \ peer\ \$\ x\ c \\ Cast_{fail} & \equiv & typeof(e) \not\preceq T \ \vee \ \neg peer\ \$\ x\ c \end{array}$$

Specification Universe information occuring in specifications as invariants, pre- and postconditions by using the owner field are transformed in the γ-function into applications of the *owner* function. For example an expression `x.h.owner == y.owner` is transformed to $owner\ \$(x.h) = owner\ y$.

This solution satisfies all stated requirements since it does not loose any available ownership information and no unnecessary proof obligations are generated. Additionally the changes to the current JIVE system are not very big. A disadvantage of this solution is that ownership information of locals and of parameters is not available to JIVE while performing the verification calculus. The verification calculus can still be performed by usage of strengthening and weakening, but it might be confusing if produced implications seem unprovable in JIVE while in Isabelle they are provable due to the added information at export.

6.2 Axiomatization of the Ownership Model

In order to enable Isabelle (and hence JIVE) to reason about ownership structures, Isabelle must be aware of the properties of the ownership model. For example, how should Isabelle know if x is the owner of y that $x \neq y$? Therefore the formalization presented in the previous section is used to axiomatize the ownership model in Isabelle.

Since ownership contexts form a tree with the root context as root, the ownership relation of all alive objects form a tree with root the *nullV* value. This tree must be axiomatized. Hence axioms for the *owner* function are to be declared stating the *owner* function to form a tree. Some properties of a tree:

1. Every node has got exactly one father, except for the root node having no father.
2. There are no cycles, meaning the father of a node can not be a descendant of that node.
3. The root node is an ancestor of every node.

Property one is guaranteed by the *owner* function, since a function always maps a specific element of the domain space to exactly one element of the image space. For *nullV*, which is the root node, the *owner* function is simply not defined.

Some additional functions are needed to express transitivity of ownership, the most important of which is *owner_n* of type $Value \rightarrow nat \rightarrow Value$ returning the value receiving after $n+1$ applications of the *owner* function to the parameter value[17]. *owner_n*'s definition in Isabelle as primrec:

[17]The reason why *owner_n* is not defined to be n applications of *owner* is that it would make many proofs more complicated since 0 would always be an exception of the rule to prove. Furthermore inductions would always have to start from 1. Therefore the definition of *owner_n* xs $0 =$ *owner* xs simplifies many proofs and inductions.

$$\begin{aligned} owner\_n\ X\ 0 &= owner\ X \\ owner\_n\ X\ (Suc\ n) &= owner\ (owner\_n\ X\ n) \end{aligned}$$

The function $is\_trans\_owner\_of$ with type $Value \rightarrow Value \rightarrow bool$ is defined as

$$is\_trans\_owner\_of\ X\ Y \quad \equiv \quad \exists\, n :: nat\ .\ owner\_n\ Y\ n\ = X$$

stating that X is a transitive owner of Y if a natural number n exists so that $owner\_n\ Y\ n = X$.

Finally $isRelevantTo$ with type $Value \rightarrow Value \rightarrow bool$ is defined as

$$isRelevantTo\ X\ Y \quad \equiv \quad is\_trans\_owner\_of\ (owner\ Y)\ X$$

meaning that a value X is relevant to a value Y if the owner of Y is a transitive owner of X.

Property two, the acyclicity of the tree, is stated as the following axiom

$$owner\_n\ X\ n \neq X$$

Alternatively is could have been axiomatized using an induction. Hence the basis and the step would have been formulated as Isabelle axioms

$$\begin{aligned} owner\ X &\neq X \\ owner\_n\ X\ n \neq X &\Longrightarrow owner\_n\ X\ (Suc\ n) \neq X \end{aligned}$$

with the add of which the acyclicity lemma could have been proven.

Property three is stated as

$$X \neq nullV \implies is\_trans\_owner\_of\ nullV\ X$$

Several other lemmas about properties of the ownership model are proven using these axioms and definitions. Some of them can be found in Appendix B. All these definitions, axioms and lemmas are part of the program-independent Isabelle theories.

7 Integrating Ownership Proof Technique into Jive

The remaining task in order to integrate the ownership proof technique into Jive is to design the modifications of those parts of current Jive that need to be changed.

7.1 Invariant Formalization

First of all the new invariant semantics of section 5.2 must be reflected in the translation of invariants to first-order logic formulas.

Again the example of a method $m()$ in class C executing in context Ω is taken. All objects outside Ω might be executing and therefore they might have broken invariants and cannot be considered. Hence they must be excluded in the invariant formalization. In section 2.4.2 the function $InvTi$ for the invariant of class Ti was defined as

$$InvTi(X, OS) \equiv typeof(X) \preceq Ti \implies I_{Ti}[X/this, OS/\$]$$

The function INV quantifying over all alive and non-null values was defined as

$$INV(OS) \equiv \forall X\ .\ (alive(X, OS) \wedge X \neq nullV \implies InvTi(X, OS) \wedge ... \wedge InvTn(X, OS))$$

This definition of INV does not fit with the modular ownership proof technique since it quantifies over all values, which could have broken invariants due to the new invariant semantics. Instead, a new INV with type $Store \rightarrow Value \rightarrow bool$

is defined taking a single value and enforcing its invariant by conjoining all type-guarded invariants of all known types.

$$INV(OS, X) \quad \equiv \quad alive(X, OS) \ \wedge \ X \neq nullV \ \Longrightarrow InvTi(X) \ \wedge ... \wedge \ InvTn(X)$$

Since objects outside Ω are not considered and the invariants of all objects inside Ω must hold upon the beginning of a function $c.m()$ with $m() \neq init()$ (point 1 of the ownership proof technique), a function quantifying over all objects inside Ω should be defined. INV_{Rel} of type $Store \rightarrow Value \rightarrow bool$ satisfies this need and is

$$INV_{Rel}(OS, X) \quad \equiv \quad \forall\, Y \ . \ (isRelevantTo(Y, X) \ \Longrightarrow INV(OS, Y))$$

where $isRelevantTo$ is the function described in section 6.2.

To satisfy point 2 of the ownership proof technique a special versions of INV_{Rel} for $init()$ methods is declared, namely $INVC_{Rel}$ of type $Store \rightarrow Value \rightarrow bool$ with the definition

$$INVC_{Rel}(OS, X) \quad \equiv \quad \forall\, Y \ . \ (X \neq Y \ \wedge \ isRelevantTo(Y, X) \ \Longrightarrow INV(OS, Y))$$

where the invariants of all objects have to be relevant to can have broken invariants.

All functions besides the $InvTi$ group and INV are program-independent.

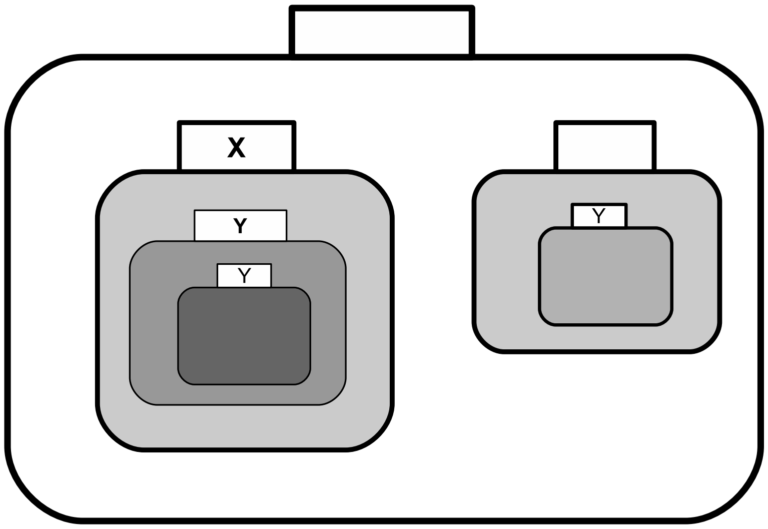
It can be easily seen that the following rules hold.

Rule 1 (INV rules)

$$\begin{array}{rcl} INVC_{Rel}(OS, X) \ \wedge \ INV(OS, X) & = & INV_{Rel}(OS, X) \\ INVC_{Rel}(OS, X) \ \wedge \ isRelevantTo\ Y\ X \ \wedge \ \neg isRelevantTo\ X\ Y & \Longrightarrow & INV_{Rel}(OS, Y) \end{array}$$

For an illustration of the second rule see figure 15 .

Figure 15: Second INV rule



7.2 Restrictions

While developing the ownership proof technique, several restrictions were imposed on the input Diet Java Card program. For these restrictions compile time checks should be added to Jive. These restrictions are:

- No static invariants or represents clauses.
- Admissibility check of invariants and represents clauses.
- No static field access (unless constant) or method call in invariants or represents clauses[18].
- No field update on a target object others than *this*.

The restriction that readonly calls are only allowed on relevant objects can only be checked as a part of the verification process but not earlier as a static compiler check.

[18]Actually already covered by the definition of admissibility.

7.3 Method Triples and Invocation Rule

7.3.1 Direct Solution

The straightforward solution is to integrate the ownership proof technique as defined at the end of section 5.6. Recall that the ownership proof technique is using the new notation:

1. At the beginning of $this.m()$ it can be assumed that $INV_{Rel}(\$, this)$ holds.
2. At the beginning of $this.init()$ it can be assumed that $INVC_{Rel}(\$, this)$ holds.
3. At the beginning of a method invocation $o.p()$ with o of universe type peer $INV(\$, this)$ must be re-established.
4. At the beginning of a method invocation $o.p()$ with o of universe type rep nothing must be assured.
5. At the beginning of a method invocation $o.p()$ with o of universe type readonly $INV(\$, this)$ must be re-established if o is in the same context as c. Additionally it must be checked that o is relevant to c.
6. After a method invocation $o.p()$ it can be assumed that $INV_{Rel}(\$, o)$ holds, meaning that if o is peer c's invariants hold.
7. At the end of $c.m()$ $INV(\$, this)$ must be re-established.

Hoare triple To satisfy point 1 and 2 of the ownership proof technique the Hoare prestate condition, which was

$$\{INV(\$) \ \wedge \ P\}$$

is now

$$\{INV_{Rel}(\$, this) \ \wedge \ P\}$$

and for a $init()$ method it is

$$\{INVC_{Rel}(\$, this) \ \wedge \ P\}$$

Point 7 alters the Hoare poststate condition from

$$\{INV(\$) \ \wedge \ P\}$$

to

$$\{INV(\$, this) \ \wedge \ P\}$$

The new Hoare triple is now

$$\{INV_{Rel}(\$, this) \ \wedge \ P\} \ T :: m(par) \ \{INV(\$, this) \ \wedge \ Q\}$$

for a regular method, and for a constructor-like $init()$ method it is

$$\{INVC_{Rel}(\$, this) \ \wedge \ P\} \ T : init(par) \ \{INV(\$, this) \ \wedge \ Q\}$$

Note that the following rules are not the final solution, they are just the straightforward integration of the ownership proof technique into JIVE.

Rule of Non-Void Invocation Point 3 to 6 request a change in the invoc-rule. First the non-void invoc-rule is considered. In current JIVE it is

$$\frac{\{Pre\} \ C :: m(par) \ \{Post\}}{\{c \neq nullV \wedge Pre[c/this, e/par]\} \ x = c.m(e) \ \{Post[x/resV]\}}$$

To comply to the ownership proof technique, the invoc-rule must be changed to

$$\frac{\{I_P \wedge P\} \ C : m(par) \ \{I_Q \wedge Q\}}{\{c \neq nullV \wedge P[c/this, e/par] \wedge I_{Pre}\} \ x = c.m(e) \ \{(I_{Post} \wedge Q)[x/resV, c/this] \wedge Own\}}$$

where I_{Post} and I_Q are defined as

$$\begin{aligned} I_{Post} &\equiv INV_{Rel}(\$, this) \\ I_Q &\equiv INV(\$, this) \end{aligned}$$

I_{Pre} is defined differently for peer, rep and readonly calls.

$$\begin{aligned} I_{Pre\_peer} &\equiv INV(\$, this) \\ I_{Pre\_rep} &\equiv true \\ I_{Pre\_readonly} &\equiv (owner\ c\ = owner\ this\ \Rightarrow INV(\$, this))\ \wedge\ isRelevantTo(c, this) \end{aligned}$$

I_{Pre_peer} emerges from point 3, I_{Pre_rep} comes from point 4 and $I_{Pre\_readonly}$ is the translation of point 5. Point 6 is reflected in the conjunction of I_P to the defined (invariant-reduced) poststate condition Q.

Own represents the ownership information descibed on page 55. If x is of universe type readonly, it is defined for a return value of universe type rep resp. peer as

$$\begin{aligned} Own_{rep} &\equiv rep\ st.typeof(this)\ \$\ x\ c \\ Own_{peer} &\equiv peer\ \$\ x\ c \end{aligned}$$

For all other cases Own is just $True$ and can be omitted.

Note that no special constructor version is needed, as I_P can be either $INV_{Rel}(\$, this)$ or $INVC_{Rel}(\$, this)$.

Rule of Void Invocation The rule of void invocation, which was

$$\frac{\{Pre\}\ C : m(par)\ \{Post\}}{\{c \neq nullV \wedge Pre[c/this, e/par]\}\ c.m(e)\ \{Post\}}$$

The ownership proof technique versions is

$$\frac{\{I_P \wedge P\}\ C:m(par)\ \{I_Q \wedge Q\}}{\{c \neq nullV \wedge P[c/this, e/par] \wedge I_{Pre}\}\ c.m(e)\ \{(I_{Post} \wedge Q)[c/this]\}}$$

where all predicates are defined as for the non-void invoc rule. Again no special rule for constructors is needed.

The problem with this direct approach is that the rule is not just applied on the complete prestate and poststate condition, as in current JIVE where the pre-respectively the poststate conditions are just substituted and conjoined to some terms. The new rules need to split the prestate and the prestate condition and deal with each part in a different way.

In current JIVE the invariant parts are conjoined in the front-end to the prestate and the poststate condition and it is very difficult to split them up in the back-end when the different invoc-rules are applied. Therefore an integration of the ownership proof technique where the conjoined prestate and poststate condition do not have to be split up would fit much smoother into the current design of JIVE. Let's call the following solution the compact solution.

7.3.2 Ownership Rule

In order to be able to make the integrated technique compact, an important property of the ownership proof technique will be used to strengthen the proof obligations for compactness purposes. To overcome the fact that stronger proof obligations mean that properties actually known must again be proven unnecessarily, an additional rule will be added to JIVE's rules enabling JIVE to prevent these unnecessary proof efforts.

The important property of the ownership proof technique is observation 1 in section 5.6, namely that no invariant of an object inside the current context can have a broken invariant except for the *this* object and for a newly created object before calling its *init*() method. Formally this property states that $INVC_{Rel}(\$, this)$ in case of no newly created object always hold during the execution of the method in question. In case of a newly created object $INVC_{Rel}(\$, this)$ holds if \$ is the store before the creation of the new object.

The new rule stating this property is the so called ownership rule saying that $INVC_{Rel}(\$, this)$ holds after a sequence of comprefs, if $INVC_{Rel}(\$, this)$ holds before the compref sequence and if the last statement in this compref sequence is not a new object creation.

$$\frac{\{INVC_{Rel}(\$, this)\ \wedge\ P\}\ cr\_last\_not\_new\{Q\}}{\{INVC_{Rel}(\$, this)\ \wedge\ P\}\ cr\_last\_not\_new\ \{INVC_{Rel}(\$, this)\ \wedge\ Q\}}$$

If the last compref is a new object creation, the ownership rule should be applied to the compref sequence but without the last new object creation. Then the prestate condition of the $init()$ call, which must follow the new object creation, can be derived using the new axiom together with Isabelle lemmas concerning stores.

7.3.3 Compact Solution

Now a new strengthened version of the ownership proof technique can be formulated which can be derived from the original version by the aid of the ownership rule. The new strengthened and (nearly) symmetric version says that:

1. At the beginning of $c.m()$ it can be assumed that $INV_{Rel}(\$, c)$ holds.
2. At the beginning of $c.init()$ it can be assumed that $INVC_{Rel}(\$, c)$ holds.
3. At the beginning of a method invocation $o.p()$ $INV_{Rel}(\$, o)$ must be assured to hold.
4. At the beginning of a method invocation $o.init()$ $INVC_{Rel}(\$, o)$ must be assured to hold.
5. At the beginning of a method invocation $o.p()$ with o of universe type readonly additionally to point 3 $isRelevantTo\ o\ c$ must be checked.[19]
6. After a method invocation $o.p()$ it can be assumed that $INV_{Rel}(\$, o)$ holds.
7. At the end of $c.m()$ the $INV_{Rel}(\$, c)$ must be assured to hold.

Now the derivation from original version is given. The ownership rule together with the INV rules at the end of section 7.1 are used for this derivation.

- $1 \Longrightarrow 1$: trivial
- $2 \Longrightarrow 2$: trivial

[19]That is the non-symmetric part.

- 3$\Longrightarrow$3: $INV(\$,c) \Longrightarrow INV_{Rel}(\$,o)$. Since o is peer, $owner\ c = owner\ o$ holds and leads to $INV_{Rel}(\$,o) = INV_{Rel}(\$,c)$. $INV(\$,c) \Longrightarrow INV_{Rel}(\$,c)$ is derived with the ownership rule and the first INV rule.
- 4$\Longrightarrow$3: $true \Longrightarrow INV_{Rel}(\$,o)$. Since o is rep, $\neg isRelevantTo\ c\ o$ holds (see subcase of lemma_twenty in Appendix B). Hence $\neg isRelevantTo\ c\ o \Longrightarrow INV_{Rel}(\$,o)$ is derived with the ownership rule and the second INV rule.
- 5$\Longrightarrow$3: $(owner\ o\ =\ owner\ c\ \rightarrow INV(\$,c))\ \wedge\ isRelevantTo\ o\ c \Longrightarrow INV_{Rel}(\$,o)$. If $owner\ c = owner\ o$ see two points before. If $owner\ c \neq owner\ o$, $\neg isRelevantTo\ c\ o$ must hold because of $isRelevantTo\ o\ c$. Now see previous point.
- 3$\Longrightarrow$4, 4 $\Longrightarrow$4 and 5 $\Longrightarrow$4: As $INV_{Rel}(\$,o)$ implies $INVC_{Rel}(\$,o)$, see previous points for a proof.
- 5$\Longrightarrow$5: trivial
- 6$\Longrightarrow$6: trivial
- 7$\Longrightarrow$7: $INV(\$,c) \Longrightarrow INV_{Rel}(\$,c)$. Directly derived with the ownership rule and the first INV rule.

The correspondingly modified Hoare triples and invoc-rules are as follows.

Hoare triples The corresponding Hoare triple is

$$\{INV_{Rel}(\$,this)\ \wedge\ P\}T : m(par)\ \{INV_{Rel}(\$,this)\ \wedge\ Q\}$$

for a regular method, and for a constructor-like *init*() method it is

$$\{INVC_{Rel}(\$,this)\ \wedge\ P\}\ T : init(par)\ \{INV_{Rel}(\$,this)\ \wedge\ Q\}$$

Rule of Non-Void Invocation The non-void invoc-rule of the compact version is split into six different subrules.

Subrule 1

$$\frac{\{Pre\}\ C : m(par)\ \{Post\}}{\{c \neq nullV \wedge Pre[c/this, e/par]\}\ x = c.m(e)\ \{Post[x/resV, c/this]\}}$$

Subrule 2

$$\frac{\{Pre\}\ C : m(par)\ \{Post\}}{\{c \neq nullV \wedge IR \wedge Pre[c/this, e/par]\}\ x = c.m(e)\ \{Post[x/resV, c/this]\}}$$

Subrule 3

$$\frac{\{Pre\}\ C : m(par)\ \{Post\}}{\{c \neq nullV \wedge Pre[c/this, e/par]\}x = c.m(e)\{Post[x/resV, c/this] \wedge rep\ ST(this)\ \$\ x\ c\}}$$

Subrule 4

$$\frac{\{Pre\}\ C : m(par)\ \{Post\}}{\{c \neq nullV \wedge IR \wedge Pre[c/this, e/par]\ x = c.m(e)\{Post[x/resV, c/this] \wedge rep\ ST(this)\ \$\ x\ c\}}$$

Subrule 5

$$\frac{\{Pre\}\ C : m(par)\ \{Post\}}{\{c \neq nullV \wedge Pre[c/this, e/par]\}\ x = c.m(e)\ \{Post[x/resV, c/this]\ \wedge\ peer\ \$\ x\ c\}}$$

Subrule 6

$$\frac{\{Pre\}\ C : m(par)\ \{Post\}}{\{c \neq nullV \wedge IR \wedge Pre[c/this, e/par]\}\ x = c.m(e)\ \{Post[x/resV, c/this]\ \wedge\ peer\ \$\ x\ c\}}$$

where IR is a short-hand for $isRelevantTo\ c\ this$ and $ST(this)$ means $st.typeof(this)$. The following table explains the usage of each subrule.

	c is peer or rep	c is readonly
x is peer or rep	Subrule 1	Subrule 2
x is readonly, m's return type is readonly	Subrule 1	Subrule 2
x is readonly, m's return type is rep	Subrule 3	Subrule 4
x is readonly, m's return type is peer	Subrule 5	Subrule 6

Rule of Non-Void Invocation The compact version of the void invoc-rule is

$$\frac{\{Pre\}\ void\ C:m(par)\ \{Post\}}{\{c \neq nullV \wedge Pre[c/this, e/par]\}\ c.m(e)\ \{Post[c/this]\}}$$

for c of universe type rep or peer, and for a readonly c

$$\frac{\{Pre\}\ void\ C:m(par)\ \{Post\}}{\{c \neq nullV \wedge IR \wedge Pre[c/this, e/par]\}\ c.m(e)\ \{Post[c/this]\}}$$

8 Examples

8.1 Layered Structures

As first example the program fragment of figure 5 should be considered. It was this example that demonstrated the weakness of current JIVE's logic. Consider figure 16 where the example of figure 5 is now coded in Diet Java Card. Additionally the proof obligations are substituted by the new ones.

Consider method *Counter.inc*(). As *Counter* and non of its supertypes declare invariants $INV(\$, this)$ is always true. Additionally the ownership rule derives $INVC_{Rel}(\$, this)$ for the poststate of the method. The proof looks like:

```
{assume INV_Rel($,this)}
{INV_Rel($,this) -> INVC_Rel($,this)} // INV rule 1
{INVC_Rel($,this) /\ S = $} // save store
void inc() {
    int t;
    {S = $}
    t = this.c;
    {S = $ /\ t = $(this.c)}
    {S = $ /\ t = S(this.c) /\ t' = t} // New logical variable
    {t' = S(this.c) /\ t' = t} // subst
    t = t + 1;
    {t' = S(this.c) /\ t = t' + 1} // subst
    this.c = t;
    {t' = S(this.c) /\ t = $(this.c) /\ t = t' + 1}
```

Figure 16: Layered Structure

```
class Counter {
    private /*@ spec_public @*/ int c;

    //@ ensures this.c == \old(this.c) + 1;
    //@ assignable c;
   {assume INV_Rel($,this)}
    void inc() {
        int t;
        t = c;
        t = t + 1;
        c = t;
    }
    {assert INV_Rel($,this)}
}

class CounterWithCache {
    private /*@ rep @*/ Counter counter;
    int cache;

    //@ invariant counter != null;
    //@ invariant cache == counter.c;

    {assume INV_Rel($,this)}
    void addOne() {
        /*@ rep @*/ Counter count;
        int cache_local;
        count = counter;

        count.inc();

        cache_local = cache;
        cache_local = cache_local + 1;
        cache = cache_local;
    }
    {assert INV_Rel($,this)}
}
```

```
        {$(this.c) = S(this.c) + 1} // subst
        {$(this.c) = S(this.c) + 1 /\ INVC_Rel($,this)} // Ownership rule
    }
    {$(this.c) = S(this.c) + 1 /\
     INVC_Rel($,this) /\ INV($,this)} // INV($,this) always true
    {$(this.c) = S(this.c) + 1 /\ INV_Rel($,this)} // INV rule 1
    {assert INV_Rel($,this) /\ $(this.c) = S(this.c) + 1}
```

Consider method *CounterWithCache.addOne*().

```
    {assume INV_Rel($,this)}
    void addOne() {

        /*@ rep @*/ Counter count;

        int cache_local;

        {INV_Rel($,this)}                          // no action until here

        count = this.counter;

        {INV_Rel($,this) /\ count = $(this.counter)}

        {INV_Rel($,this) /\ count = $(this.counter)
         /\ $($(this.counter).c) = $(this.cache)      // from INV_Rel($,this)
         /\ $(this.counter) != nullV}                 // from INV_Rel($,this)

        {count = $(this.counter) != nullV
         /\ $($(this.counter).c) = $(this.cache)
         /\ INV_Rel($,count)}                // INV rule 2 since count is rep

        {INV_Rel($,count) /\ $ = OS'                       // save $ to OS'
          /\ oldcount' = OS'(count.c)                 // New logical variable
          /\ count = $(this.counter) != nullV
          /\ $($(this.counter).c) = $(this.cache)}

        {INV_Rel($,count) /\ $ = OS' /\ oldcount' = OS'(count.c)
         /\ count = $(this.counter) != nullV
         /\ oldcount' = $(this.cache)} // subst

        count.inc();
```

```
{INV_Rel($,count) /\ $(count.c) = OS'(count.c) + 1  // Invoc-rule
 /\ count = OS(this.counter) != nullV            // Invoc-var-rule
 /\ oldcount' = OS(this.cache)                // Invoc-var-rule
 /\ OS(this.counter) = $(this.counter)        // Invoc-rule
 /\ OS(this.cache) = $(this.cache)}            // Invoc-rule

{INV_Rel($,count) /\ $(count.c) = oldcount' + 1     // subst
 /\ count = $(this.counter)                    // subst
 /\ oldcount' = $(this.cache)                  // subst
 /\ $(this.counter) != nullV}               // subst

{$($(this.counter).c) = $(this.cache) + 1   // substs and weakening
 /\ $(this.counter) != nullV}

cache_local = this.cache;

{$($(this.counter).c) = $(this.cache) + 1 /\ cache' = $(this.cache)
 /\ cache' = cache_local                     // New logical variable
 /\ $(this.counter) != nullV}

{$($(this.counter).c) = cache' + 1 /\ cache' = cache_local // subst
 /\ $(this.counter) != nullV}

cache_local = cache_local + 1;

{$($(this.counter).c) = cache' + 1 /\ cache_local = cache' + 1
 /\ $(this.counter) != nullV}

{$($(this.counter).c) = cache_local                   // subst
 /\ $(this.counter) != nullV}

this.cache = cache_local;

{$($(this.counter).c) = cache_local /\ $(this.cache) = cache_local
 /\ $(this.counter) != nullV}
// as this.counter != this.cache && this != this.counter (type info)

{$($(this.counter).c) = $(this.cache)                     // subst
 /\ $(this.counter) != nullV}
```

```
        {INV($,this)} // thats CounterWithCache's invariant

        {INV($,this) /\ INVC_Rel($,this)}                  // ownership rule

        {INV_Rel($,this)} // INV rule 1
     }
     {assert INV_Rel($,this)}
```

Hence the compact solution can successfully prove layered object structures !

8.2 Universe Cast

Consider the following methods appearing in class T:

```
    /*@ pure rep @*/ Object repObj() {
        /*@ rep @*/ Object temp;
        temp = new /*@ rep @*/ Object;
        return temp;
    }
    //@ signals (Exception) false;
    {assume INV_Rel($,this)}
    /*@ pure @*/ void testCallCast() {
        /*@ readonly @*/ Object readonlyO;
        /*@ rep @*/ Object repO;
        readonlyO = this.repObj();
        realrepR = (/*@ rep @*/ Object) repR;
    }
    {assure INV_Rel($,this)}
```

The interesting part is that the cast is proven not to throw an exception.

```
    //@ signals (Exception) false;
    {assume INV_Rel($,this)}
    /*@ pure @*/ void testCallCast() {
        /*@ readonly @*/ Object readonlyO;
        /*@ rep @*/ Object repO;
        {INV_Rel($,this)}                  // no action until here
        readonlyO = this.repObj();
        {INV_Rel($,this) /\ rep T $ readonlyO this} // invoc-rule
        {INV_Rel($,this) /\ rep T $ readonlyO this /\
         type repR <: CClassT Object}               // type info
        // cast will pass !!
        realrepR = (/*@ rep @*/ Object) repR;
        {INV_Rel($,this)}    // cast axiom
    }
    {assure INV_Rel($,this)}
```

8.3 Init

Consider the following code occuring in a class T

```
    void init() {...}

    void foo() {
        /*@ rep @*/ T t;
        ...
        t = new /*@ rep @*/ T();
        t.init();
        ...
    }
```

The problem is how to verify the *init*() call. This time the proof is done backwards.

```
    {assure INV_Rel($,this)}
    }
        ...
        {INVC_Rel($,this)}     // ownership rule (forward)
        {INV_Rel($,t)}         // weakening by INV rule 2 since t is rep
        t.init();
        {INVC_Rel($,t)}            // invoc-rule
        t = new /*@ rep @*/ T();
        {INVC_Rel($<T>, new($,T))} // new axiom
        {INVC_Rel($,this) -> INVC_Rel($<T>,new($,T))} // weakening (see below)
        {INVC_Rel($,this)}         // ownership rule (forward)
        ...
    /*@ rep @*/ T t;
void foo() {
{assume INV_Rel($,this)}
```

The question is how $INVC_{Rel}(\$\langle T\rangle, new(\$, T))$ is derived from $INVC_{Rel}(\$, this)$.

The derivation is done as follows. INV rule 2 states that for an object o inside but not element of *this*' context $INVC_{Rel}(\$, this) \implies INV_{Rel}(\$, o)$ holds. $new(\$, T)$ is inside but not element of *this*' context as it is a rep. Therefore let's assume that o and $new(\$, T)$ will be elements of the same context.

Since no location in $ depends on $new(\$, T)$ as it is not alive, the invariants of all objects relevant to o and alive in $ must still hold in store $\$\langle T\rangle$. Hence $INVC_{Rel}(\$\langle T\rangle, new(\$, T))$ holds as $new(\$, T)$'s invariant are excluded in $INVC_{Rel}(\$\langle T\rangle, new(\$, T))$.

8.4 Universe Types

The foo method call in method *bar* in the following code snippet of class T can not be proven without considering the information provided by universe types.

```
//@ requires a != b;
abstract void foo(/*@ readonly @*/ T a, /*@ readonly @*/ T b);

void bar(/*@ rep @*/ T c, /*@ peer @*/ T d) {
    foo(c,d);
    ...
}
```

Let's prove *bar*.

```
{assume INV_Rel($,this)}
void bar(/*@ rep @*/ T c, /*@ peer @*/ T d) {
    {INV_Rel($,this)}
    {INV_Rel($,this) /\ rep T $ c this
     /\ peer $ d this}         // universe type info
    {INV_Rel($,this) /\ owner c = this
     /\ owner d = owner this}  // from def of rep and peer
```

```
        {INV_Rel($,this) /\ owner c = this /\ owner d = owner this
         /\ owner this /= owner}   // lemma Ownership.basic
        {INV_Rel($,this) /\ owner c /= owner d}
        {INV_Rel($,this) /\ c /= d}
        foo(c,d);
        {INV_Rel($,this)}          // invoc-rule
        ...
    }
    {assure INV_Rel($,this)}
```

8.5 Readonly method calls

Consider the following code of class T:

```
    void bar() {...}

    // requires T.owner.owner = this;
    void foo(/*@ readonly @*/ T t) {
        t.bar();
        ...
    }
```

It is proven as follows:

```
    {INV_Rel($,this) /\ owner (owner t)) = this}
    void foo(/*@ readonly @*/ T t) {
        {INV_Rel($,this) /\ owner (owner t)) = this}
        {INV_Rel($,this) /\ owner_n (owner_n t 0) 0 = this} // def of owner_n
        {INV_Rel($,this)
         /\ owner_n t 1 = this} // rule Ownership.OwnerNFusion
        {INV_Rel($,this)
         /\ isRelevant t this} // rule Ownership.RelevantToTransitiveOwner
        {INV_Rel($,t)
         /\ isRelevant t this} // rule Ownership.INVToRelevantInv
        t.bar();
        {INV_Rel($,t)} // invoc-rule
        ...
    }
```

9 Implementation Details

As a part of this master thesis, the concepts described in earlier sections were implemented in JIVE. This section lists details about the implementation, where

- things were implemented differently than presented in earlier sections.
- parts still contain known problems.
- important design decisions were made.

Since implementation details are explained, knowledge of the implementation of JIVE is assumed.

9.1 Integrating Ownership into JIVE

The ownership version of JIVE was integrated into the non-modular version of JIVE in such a way that the user can choose which version of JIVE he wants to use by selecting the start-up option "-ownership". Internally a variable is set to determine whether the "-ownership" flag was set, and accordingly the appropriate code is executed.

For the rules the following convention was made. All changed rules were renamed, where the string "_ownerver" (for ownership version) was added after the rules name but before {forward,backward} (e.g. KEx_inst_cast_axiom is changed to KEx_inst_cast_ownerver_axiom). The additional ownership rules (backward and forward) are recognized by the string "ownership" in their names.

The new rules and axioms are:

- KEx_invoc_ownerver_forward and KEx_invoc_ownerver_backward
- KEx_invoc_void_ownerver_forward and KEx_invoc_void_ownerver_backward
- KEx_ownership_forward and KEx_ownership_backward
- KEx_inst_cast_ownerver_axiom and KEx_check_cast_ownerver_axiom

9.2 Admissibility checks

Since the admissiblity checks were inserted into the JML compiler and not into the Jive front-end specific code, all possible Java expression had to be considered. As Java supports arbitrary nesting of expressions, it was very hard to consider all possible cases of expression nesting.

The admissibility checker was implemented with the visitor pattern as stateful visitor.

Constants All fields declared as final are considered to be constant, as they can not be modified after their initialization.

Expressions It is non-trivial to extract all expressions to be checked by the admissibility checker from an invariant. For example the expression `(intfield > 0 ? peerfield : peerfield2).outerfield` contributes three expressions, namely `intfield`, `peerfield.outerfield` and `peerfield2.outerfield`. Other examples of expressions contributing several expressions to check are array accesses, new array creations, array initializers, new object creations and method calls.

Universe Down-Casts Universe down-casts occuring in an expression must be handled specially as they provide information about the expression to be casted. For example it is known for an expression casted to rep that the object the casted expression is referring to is an element of the context owned by the *this* object. Hence further expressions building on this cast can profit from this knowledge.

Let's illustrate this fact by an example expression `((rep T) repF.roF1).roF2`, where `repF` denotes a rep field and `roF1` resp. `roF2` denote readonly fields, all of type T. Without the cast this expression is not admissible, since the object referred to by `repF.roF1` might not be inside the context owned by *this*. Therefore the invariant of *this* must not depend on a field of the object referred to by `repF.roF1` and hence `repF.roF1.roF2` is not admissible. The cast provides the information that `repF.roF1` is inside the context owned by *this*, therefore the invariant of *this* is allowed to depend on fields of `repF.roF1`. Hence `((rep T) repF.roF1).roF2` is admissible.

Hidden *this* and subclass separation Subclass separation requires the left-most field access to be on a field declared in the current class. The left-most field access can be separated from *this*, e.g. *this* can be casted before the field is actually accessed. Therefore special care must be taken in the admissibility checker that subclass separation is not circumvented by trying to hide *this*.

Open problems. The current implementation of the admissibility checker has two known problems:

- Array access on a newly created array with initializer: e.g. `new Object[]{peerfield}[0]` is rejected by the admissibility checker, although it is actually admissible. This expression is equal to `peerfield`, which of course is admissible. The rejection is due to the fact that when the admissibility checker encounters an array access it requires the left-most field access to be a rep field. This requirement follows from the fact that for admissibility checking array access are equal to field accesses. Furthermore admissible expressions with the left-most field access of type other than rep may only have one field access, when not considering newly created arrays. As an array access can not be the left-most field access, the admissibility checker requires the left-most field access to be a rep field when it encounters an array access. Hence the above mentioned expression is rejected.
- Hidden *this*: e.g. `(field < 0 ? this : this).field == 0` is reject although actually admissible. The expression is admissible since it is equal to the admissible `this.field`. As the admissibility checker does not realize that the field access is the left-most one, since *this* is not the prefix of the field access, the expression is rejected.

Additional checks The following checks are part of the admissibility checks done in the JML compiler but were implemented outside the main admissibility checker classes:

- No static invariants
- No static represents clauses

9.3 Ownership formalization

As mentioned in a footnote in section 6, the *owner* function could be defined as a short-hand for an access to the *owner* field. In a early stage of this thesis the *owner* function was indeed defined that way. As a consequence the *owner* function and all other related functions as *owner_n*, *isRelevantTo* and *is_trans_owner_of* must take an additional parameter of type store in order to access the owner field for a certain value.

In the current implementation of this thesis the store parameter was removed, and hence the *owner* function has type $Value \rightarrow Value$ as described in section 6. The advantage of this solution is that ownership relations once they are defined can be used for all different stores without any proof effort. On the other hand once defined these relations could be potentially used for not alive objects, which would not make any sense. Further considerations are needed to find out whether it could even cause soundness problems.

The use of a store parameter in the *owner* function requires additional axioms for transferring an ownership relation from one store into subsequent stores, as for example

$$\begin{aligned}\forall OS :: Store\ OS' :: Store\ .\ \ &((\forall X :: Value\ .\ alive(X, OS)\ \Longrightarrow alive(X, OS'))\\ &\Longrightarrow owner\ OS\ X = owner\ OS'\ X)\end{aligned}$$

9.4 Subclass separation

Although subclass separation is fully considered in the previous sections, it was only partially implemented. The current implementation checks that rep fields, methods returning rep objects and methods taking rep objects as parameters must be declared to be private. These checks where implemented directly in the JML compiler as part of the admissibility checks. Additionally the invariant and represents clause admissibility checks are performed as described in section 5.5 including the parts contributed by subclass separation. Hence the only place where subclass separation can still be circumvented are casts, since the parts of the cast pass condition contributed by subclass separation are not yet implemented. These tests would require an additional field *declClass* of type $\backslash TYPE$ declared in *java.lang.Object*. Furthermore JIVE must be able to handle fields of type $\backslash TYPE$ correctly, as currently a field access in JIVE yields a Value, whereas types are of sort JavaType.

9.5 Minor details

This Through this thesis the variable *this* was used in formulas to refer to the receiver object of the current executing method. Lately *this* was forbidden to occur in poststate conditions, which forced the addition of a logical variable T. In the prestate condition *this* is saved to T. Afterwards T is used instead of *this*. Although the implementation was changed to use only T in poststate conditions, the formulas in this thesis were not changed.

Checks Besides admissibility and subclass separation checks the implementation checks in the JIVE front-end whether all field updates are on target *this*.

Rules In section 7.3.3 several rules were split into different subrules. In the implementation these subrules could have been integrated into one class per rule.

10 Future Tasks

Modularity In order to achieve full modularity, the semantics and the translation of assignable clauses must be changed.

A method p may modify a location f a client's model field m depends on. As m can not be mentioned in the assignable clause of p, since the client in which m is declared in is not known while writing p, the semantics of assignable clauses must be changed to permit changes in model fields of objects outside the current execution context. Lately a so-called light ownership concept was introduced into current JIVE, where a method may modify all fields not in the current context. This approach is not sufficient as information about modified fields inside the current context could be needed to verify the method in question. Additionally concrete fields of objects outside the context of the method execution can not be changed by this method.

Future work could elaborate whether the requirement on objects to specify on what data groups their model fields may depend would help. Additionally it must be worked out what sort of constraints on the members of data groups are useful. Possible constraints are for example context restrictions and disjunction of data groups.

Additionally the current downward closure computation requires the knowledge of all subclasses, which renders this computation non-modular. In order to achieve modularity, a technique must be developed where methods calling virtual methods can be verified without the knowledge of all subclasses of the classes where these virtual methods are declared, but still permitting future subclasses to add locations to their state. Maybe the introduction of residues as described in [9] could help.

Proof Technique Ownership proof technique and its admissibility condition does not allow invariants to depend on the state of objects in the same context. Mutually dependent object structures can hence not be verified since their invariants are not ownership admissible. To solve this problem the visibility technique [3, Section 9], which was already metioned in section 5, could be used. The visibility technique supports mutually dependent object structures by weakening its admissibility condition, but its proof obligations are more diffcult to prove. That is the trade-off mentioned in the introduction, that a weaker admissibility condition normally emerges in harder to prove proof obligations and vice versa.

Arrays As remarked in section 2.1.1, JIVE was planned to support Diet Java Card programs, but currently only Java-KEx programs are fully supported. The contribution of this thesis are valid for both target languages, Diet Java Card and Java-KEx, with one exception. Arrays were not considered in the formalization of the ownership model. When JIVE will support full Diet Java Card the implications on the contribution of this thesis emerging from the use of the universe type system in connection with arrays should be further elaborated.

For example the cast-rule should have a special version for array casts to check the second universe type. For an array `arr` casted to universe type `rep peer` the checked term would be:

$$\begin{aligned} &arr = null \;\vee\; (owner\ arr = this \;\wedge\; \$(arr.declClass) = st.typeof(this) \\ &\wedge\; \forall n :: nat\ .\ (0\ \leq n \;\wedge\; n < \$(arr.length) \;\Longrightarrow peer\ \$\ \$(arr[n])\ arr)) \end{aligned}$$

Tactics JIVE is equipped with automatic proof tactics, one of which is a practical weakest precondition calculus tactic. Although in ownership JIVE the ownership versions of the invoc-rule, the invoc-void-rule and the cast-axiom are used instead of their non-ownership equivalents in the practical weakest precondition tactic, further considerations are needed whether this simple substitution of the rules is sensible. Additionally the ownership rule is not yet integrated into the practical weakest precondition tactic. Since the ownership rule is heavily needed in the verification calculus, the practical weakest precondition tactic of the current version of ownership JIVE will not be able to prove most programs automatically. Therefore the integration of the ownership rule into the practical weakest precondition tactic is an important future task. For example the practical weakest precondition tactic could for example apply the ownership rule before every method invocation statement.

Isabelle Since a first version of the program module exporting the JIVE-generated first-order logic implications to Isabelle was ready just before the end of this thesis, there are no results about the actual difficulty of proving these implications in Isabelle. This applies to the JIVE-generated implications in general and to the parts dealing with ownership-specific properties in particular. Hence future work could investigate on the repertoire of axioms and lemmas provided by the Ownership theory, whether they are useful or not, whether important lemmas or even axioms are missing and how these axioms and lemmas are normally used in proving the exported lemmas.

11 Conclusion

This thesis introduced current JIVE by describing its architecture and its target language, Diet Java Card together with JML. While explaining invariants the invariant semantics of JIVE was emphasised, which requires the invariants of all objects to hold at points of execution control transfer.

The basic elements of the formalization of the object store model of Poetzsch-Heffter and Müller's program logic were explained. The JML transformation to first-order logic formulas was then descibed, where current JIVE's invariant transformation was especially pointed out showing its non-modularity. Finally those rules and axioms of JIVE's logic relevant to this thesis were introduced.

As a next step it was shown that current JIVE's proof technique is not able to handle layered object structures since the used invariant semantics does not permit objects to break the invariants of their clients. Therefore a first step was taken to weaken the invariant semantics allowing objects to break the invariants of their clients but causing the new problem of re-entrant method calls on objects with broken invariants.

This problem can be solved using the universe type system. A new invariant semantics was defined, weaker than current JIVE's but stronger than the first step's. This new invariant semantics solves the problem of re-entrant method calls. But if the invariant semantics was taken as proof obligation it would still be non-modular.

It was then shown that if additional restrictions are imposed on the input program the universe invariant guarantees that only the invariant of the currently executing object can be broken by the currently executing method, therefore $INVC_{Rel}(\$, this)$ always holds. The most important restriction is the requirement of ownership admissible invariants and represents clauses which restricts the locations they may depend on. Eventually the modular ownership proof technique was formulated.

Since the ownership proof technique is based on the universe type system, JIVE must be aware of the properties of the ownership model. Several possible solutions to formalize ownership were discussed. The chosen formalization uses the *owner* function to represent ownership. The approach how this function is used to preserve all available ownership information without generating new proof obligations was explained in detail.

Eventually the integration of the ownership proof technique into JIVE was decribed by first showing how invariants should be transformed into first-order logic formulas according to the new invariant semantics. The straightforward solution of integrating the ownership proof technique into JIVE is not feasible as it requires JIVE to split in the back-end the conjoined prestate resp. poststate condition.

A compact solution was then developed not lacking this draw-back. This compact solution requires an additional Hoare rule, the so-called ownership rule, stating the if $INVC_{Rel}(\$, this)$ holds at some point of execution in a method $m()$ it will hold at any subsequent point (if it is not a point of new object creation). It was proven that the compact solution can be derived from the original one together with the ownership rule.

Some examples showed that the compact solution is indeed capable of proving layered object structures. Additionally universe casts and ownership specifications are successfully handled by the compact solution.

To improve the current implementation future work should consider arrays, JIVE's tactics handling ownership and get experience in actually proving the exported implications in Isabelle using the provided axioms and lemmas.

An other approach to solve the problem of proving layered structures is taken by the Boogie tool [18]. It tries to solve the problem by an explicit treatment in specification when invariants have to hold. In the Boogie technique, an object is either in a valid or a mutable state. An object has to satisfy its invariants only when it is in a valid state, and only objects in a mutable state can be modified. This state information is represented by a specification-only field. This field can be modified through two special statements, `pack` and `unpack`. An ownership structure is used similar to the one of the universe type system. As in the ownership proof technique the Boogie technique enforces that all (transitive) owner objects of a mutable object are also mutable allowing the mutable object to break the invariants of its (transitive) owner objects. The invariants are checked when the objects are packed, i.e. turned from mutable to valid. The `pack` and `unpack` statements significantly increase the specification overhead, as it is difficult to figure out where to use these two statements. Additionally method specifications become more complex as it is needed to describe what objects are valid. For the Boogie methodology see [20].

Future work should integrate the visibility technique into JIVE in order to handle mutually dependent object structures. Additionally one should investigate the implications of the universe type system on modifies clauses, in order to achieve full modularity.

12 Acknowledgements

I would like to thank all people who have helped me realizing this master project. First of all my supervisors, Prof. Peter Müller and Adam Darvas for their guidance throughout this project, for their useful comments and criticism and for their time they spent helping me doing this project in a correct way. Without Erich Laube's testcases the admissibility checks would still be erroneous. Finally I want to special thank my dear wife and my dear parents for their continuous support.

A Ownership Relationship Notation

The following table lists the notation for different ownership relations between objects and contexts. While Roman letters denote objects, Greek letters mean contexts.

X	is in the same context as	Y	owner of X equals the owner of Y
X	is an element of the context of	Y	owner of X equals the owner of Y
X	is inside the context of	Y	owner of Y is a transitive the owner of X
X	(directly) owns	Y	X is the owner of Y
X	transitively owns	Y	X is a transitive owner of Y
X	is owned by	Y	Y is the owner of X
X	is inside the context owned by	Y	Y is a transitive owner of X
X	is an element/member of	Ω	owner of Ω equals the owner of X
X	is inside	Ω	owner of Ω is a transitive owner of X
X	is in	Ω	(never used due to possible confusion)
X	(directly) owns	Ω	X is owner of all members of Ω
X	transitively owns	Ω	X is transitive owner of all members of Ω
Ω	is the same as	Ψ	owner of of Ω equals the owner of Ψ
Ω	is inside	Ψ	owner of Ψ is a transitive owner of Ω
Ω	is a descendant of	Ψ	a transitive owner of Ω is a member of Ψ
	The owner of	Ω	the owner of all elements of Ω
	Context of	X	all objects with owner equals owner of X
Ω	is the context of	X	owner of Ω equals the owner of X
X	is owned by	Ω	owner of X is a member of Ω
Ω	is a child of	Ψ	owner of Ω is a member of Ψ
Ω	is the owner of	Ψ	owner of Ψ is a member of Ω

Negations are analogous, just replace *is* by *is not, inside* by *outside* or add *does not.* As a simple rule: *X is element of* or *X is in the same* refer to exactly one context, wereas *X is inside* is transitiv and several contexts can be meant. *In* is not used since it is confusing whether X is in Y's context means X is inside the context of Y, X is in the same context as Y or X is in the context owned by Y. *In* could actually be defined here but since it is probably used in other papers with a different semantics, the reader will be confused.

B Selected Ownership Lemmas

These definitons, axioms and lemmas are program-independent and taken from the Ownership theory.

B.1 Definitions

owner_n

$$\begin{aligned} owner\_n\ X\ 0 &= owner\ X \\ owner\_n\ X\ (Suc\ n) &= owner\ (owner\_n\ X\ n) \end{aligned}$$

is_trans_owner_of

$$is\_trans\_owner\_of\ X\ Y \quad \equiv \quad \exists\, n :: nat\ .\ owner\_n\ Y\ n\ = X$$

isRelevantTo

$$isRelevantTo\ X\ Y \quad \equiv \quad is\_trans\_owner\_of\ (owner\ Y)\ X$$

B.2 Axioms

Acyclic

$$owner\_n\ X\ n \neq X$$

B.3 Lemmas

Basic This lemma states the non-reflexitivity of the ownership function. It is called basic since it is the most basic ownership property.

$$owner\ X \quad \neq \quad X$$

Induction This rule is the induction step of acyclicity

$$owner\_n\ X\ n \neq X \quad \Longrightarrow \quad owner\_n\ X\ (Suc\ n) \neq X$$

Acyclic2

Acyclicity defined by the is_trans_owner_of function

$$\neg is\_trans\_owner\_of\ X\ X$$

OwnerIsNotTransitivelyOwned The following lemma states that the owner of an object is not transitively owned

$$X = owner\ Y \Longrightarrow \neg is\_trans\_owner\_of\ Y\ X$$

NonReflexive This lemma states the non-reflexivity of owner and is_trans_owner_of for two equal values

$$\begin{aligned} X = Y \quad \Longrightarrow \quad & X \neq owner\ Y\ \wedge\ Y \neq owner\ X \\ & \wedge \neg is\_trans\_owner\_of\ Y\ X \\ & \wedge \neg is\_trans\_owner\_of\ X\ Y \end{aligned}$$

RelevanceReflexivity An object is relevant to itself.

$$isRelevantTo\ X\ X$$

PeerRelevance Peer objects are always relevant

$$owner\ X\ = owner\ Y \Longrightarrow isRelevantTo\ X\ Y$$

RelevantToTransitiveOwner An object is always relevant to all its transitive owners

$$owner\_n\ X\ n = Y \Longrightarrow isRelevantTo\ X\ Y$$

with the subcase RepRelevance

$$owner\ X\ = Y \Longrightarrow isRelevantTo\ X\ Y$$

OwnerNFusion Two applications of owner_n can be combined to one application. Note the $+1$ in the sum of the nat parameters. It's because $owner\_n\ X\ 0$ is not X but $owner\ X$.

$$owner\_n\ (owner\_n\ X\ n1)\ n2 = owner\_n\ X\ (n1 + n2 + 1)$$

TransitiveOwnerNotRelevance Transive owners are not relevant

$$\forall n :: nat\ .\ \neg isRelevantTo\ (owner\_n\ X\ n)\ X$$

with the subcase OwnerNotRelevance

$$\neg isRelevantTo\ (owner\ X)\ X$$

INVRelToINVCRel

$$INV\_Rel\ OS\ X \Longrightarrow INVC\_Rel\ OS\ X$$

INVRelToINV

$$INV\_Rel\ OS\ X \Longrightarrow INV\ OS\ X$$

Relevance_splitting This is INV rule 1

$$INVC\_Rel\ OS\ X\ \wedge\ INV\ OS\ X = INV\_Rel\ OS\ X$$

INVToRelevantInv (not yet proven)

$$INV\_Rel\ OS\ X\ \wedge\ isRelevantTo\ Y\ X\ \implies\ INV\_Rel\ OS\ Y$$

INV_rule_2 (not yet proven)

$$INVC\_Rel\ OS\ X \wedge isRelevantTo\ Y\ X \wedge \neg isRelevantTo\ X\ Y\ \implies\ INV\_Rel\ OS\ Y$$

References

[1] J. Meyer, P. Müller, and A. Poetzsch-Heffter. The Jive system-implementation description. 2000.

[2] Adam Darvas and Peter Müller. Formal encoding of JML Level 0 Specifications in Jive. Internal Report.

[3] Peter Müller, Arnd Poetzsch-Heffter and Gary T. Leavens. Modular Invariants for Layered Object Structures. Technical Report 434, Department of Computer Science, ETH Zürich, 2004.

[4] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. To appear in ACM SIGSOFT Software Engineering Notes. Available from '`ftp://ftp.cs.iastate.edu/pub/leavens/JML/prelimdesign.pdf`'.

[5] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller and Joseph Kiniry. JML Reference Manual. Under continuous development. Available from '`ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmlrefman.pdf`'.

[6] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):532, October 2005.

[7] Virtual Machine Specification for the Java Card Platform. Sun Microsystems, 2003. Available from '`http://java.sun.com/products/javacard/specs.html`'

[8] C.A.R.Hoare. An axiomatic basis for computer programming. Communications of the ACM, pages 576-583, October 1969.

[9] K. R. M. Leino. Data Groups: Specifying the modification of extended state. In Proceedings of the '98 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '98), volume 33(10) of ACM SIGPLAN Notices, pages 144-153, October 1998.

[10] Tobias Nipkow, Lawrence C. Paulson and Markus Wenzel. Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Springer-Verlag, 2005.

[11] David Detlefs, Greg Nelson and James B. Saxe. Simplify: A Theorem Prover for Program Checking. Technical Report, Hewlett-Packard Systems Research Center. 2004.

[12] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[13] Arnd Poetzsch-Heffter and Peter Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *European Symposium on Programming Languages and Systems (ESOP)*, volume 1576 of *Lecture Notes in Computer Science*, pages 162-176. Springer-Verlag, 1999.

[14] R. Bornat. Proving pointer programs in Hoare logic. In R. Backhouse and J. N. Oliveria, editors, Mathematics of Program Construction, Lecture Notes in Computer Science 1837, pages 102-126, 2000.

[15] Joseph M. Morris. A general axiom of assignment. Assignment and linked data structures. A proof of the Schorr-Waite algorithm. In M. Broy and G. Schmidt, editors, Theoretical Foundations of Programming Methodology, (Lecture Notes International Summer School, Markoberdorf), pages 25-51, Reidel, Dortrecht, Netherlands, 1982.

[16] Arnd Poetzsch-Heffter, Jean-Marie Gaillourdet and Nicole Rauch. Soundness and Relative Completeness of a Programming Logic for a Sequential Java Subset. Internal Report.

[17] N. Rauch, J. Schäfer and A. Wiebel. The Java Interactive Verification Environment User Handbook. Pages 9-12. Internal Report.

[18] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In CASSIS 2004, LNCS vol. 3362, Springer, 2004.

[19] Peter Müller and Arnd Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.

[20] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, Wolfram Schulte : Verification of object-oriented programs with invariants, in Journal of Object Technology, vol. 3, no. 6, June 2004, Special issue: ECOOP 2003 workshop on FTfJP, pp. 2756, '`http://www.jot.fm/issues/issue 2004 06/article2`'.

[21] Adam Darvas and Peter Müller. Reasoning About Method Calls in Interface Specifications. To appear in *Journal of Object Technology (JOT).* 2006.