Translating invariant proofs between Spec♯ and JML

Nicu G. Fruja
Computer Science Department
ETH Zürich, Switzerland
fruja@inf.ethz.ch

Peter Müller
Microsoft Research
Redmond, USA
mueller@microsoft.com

[DRAFT] Technical Report
13 December 2007

Abstract

This work aims at relating the methodologies Spec♯ and JML rely on by defining a semantics-preserving translation scheme between Spec♯ and JML. This translation shall map a Spec♯ program, verifiable in the Spec♯ methodology, to a semantic equivalent JML program, verifiable in the JML methodology, and vice versa.

1 Introduction

The Spec♯ programming system [3] is based on a methodology for specifying and verifying object-oriented programs. The basics of this sound methodology are described in [2, 4].

The methodology the Java Modeling Language (JML) [1] relies on is built on top of the *Universe Type System*. Details of this sound methodology are presented in detail in [6].

Both the Spec♯ and JML methodologies use *object invariants* to specify the consistency of data and an *ownership model* to organize objects into contexts. The two methodologies differ, however, in several respects. The Spec♯ methodology leverages a program's hierarchy of abstractions. For this purpose, the methodology tracks ownership relations, dynamically, by means of *ghost fields*, *invariants*, and new *program statements* (**pack** and **unpack**). Unlike Spec♯'s methodology, the methodology JML is based on uses the Universe Type System to statically enforce the ownership model.

In the Spec♯ methodology, one can reason separately about the object invariants declared in different subclasses. In JML, this is, however, not possible. That is because in JML an object is either in a state where all its invariants, *i.e.*, also the invariants declared in subclasses, are known to hold or in a state where all the invariants are allowed to be violated.

The two methodologies also differ with the respect to the definitions of legal assignments and admissible invariants. Thus, assignments allowed in the Spec♯ methodology are not legal in the JML methodology and vice versa. Similarly, invariants permitted in the JML methodology are not admissible in the Spec♯ methodology and vice versa.

The goal of this work is to relate the two methodologies by defining a semantics-preserving translation scheme between Spec♯ and JML. This translation is supposed to map a Spec♯ program, verifiable in the Spec♯ methodology, into a semantic equivalent JML program, verifiable in the JML methodology, and vice versa. Such a translation scheme is, for example, useful as it can enable carrying any progress in one methodology into the other methodology.

Throughout this work, to simplify the exposition, we make the following assumptions:

- Routines have only one parameter besides the `this` receiver.
- Expressions do not have side effects.
- Routines' bodies consist of assignments and method calls only.
- Classes have a single constructor.
- Constructors do not call methods except for the implicit call of the direct superclass constructor.
- There are no constant field accesses.

Outline The rest of this work is organized as follows. Section 2 defines translation schemes, from Spec♯ to JML and vice versa, for restricted methodologies that consider only individual objects. Translation schemes for extended methodologies that reason about aggregate objects are defined in Sections 3 and 4. Thus, the translation schemes for the methodologies that can check ownership-based invariants are described in Section 3, while schemes for the methodologies that can reason about visibility-based invariants are presented in Section 4.

2 Invariants of single objects

2.1 From Spec♯ to JML

Task Given a Spec♯ program whose executions are legitimate in Spec♯, we want to translate it (without changing its behavior) to a JML program whose executions are legitimate in JML.

2.1.1 Considered Spec♯ subset

[*inv*'s **restrictions**] The field *inv* cannot be mentioned in invariants and cannot be directly updated. It can only be read in method preconditions and postconditions.

Definition 1 *Let T and S be two classes such that S is the direct superclass of T. The statements* **pack** *and* **unpack** *are defined as follows:*

pack o **as** $T \equiv$
  assert $o \neq null \wedge o.inv = S$
  assert $Inv_T(o)$
  $o.inv := T$

unpack o **from** $T \equiv$
  assert $o \neq null \wedge o.inv = T$
  $o.inv := S$

[**Constructors**] Every constructor has the postcondition

$$inv = T$$

where T is the class of the constructor.

Definition 2 (Legal assignment) *If f is declared in class T, then the assignment $o.f = exp$ is* legal *iff $T < o.inv$.*

Definition 3 (Admissible invariant) *An invariant in a class T is* admissible *iff each of its access expressions is of the form* `this`*.f, where f is a field of T (not necessarily declared by T).*

Definition 4 (Legitimate Spec♯ **program execution)** *In the* Spec♯ *methodology, a program execution is* legitimate *iff each of the program's* **assert** *statements succeeds at run-time.*

2.1.2 Considered JML subset

Definition 5 (Legal assignment) *Every assignment $o.f = exp$ is* legal.

Definition 6 (Admissible invariant) *An invariant in a class T is* admissible *iff each of its access expressions is of the form* `this`*.f, where f is a field declared by T.*

Definition 7 (Legitimate JML program execution) *A program execution is* legitimate *iff each of its objects satisfies its invariant in each visible state.*

2.1.3 Translation

Idea We consider in class `Object` a *ghost* type-valued field. More precisely, we assume that `Object` declares an instance field *inv* of type `Type`.

Subset of Spec♯ We can only consider a *subset* of Spec♯ programs. Let us denote by $\text{Spec}^{\sharp}{}_{\mathcal{S}}$ this subset. Given the definition of JML admissible invariants (Definition 6), $\text{Spec}^{\sharp}{}_{\mathcal{S}}$ only allows admissible invariants which refer to fields declared by the enclosing class.

Definition 8 (Translation) *The translation function Tr is defined as follows:*

$$Tr\,(\textbf{pack}\ o\ \textbf{as}\ T) \equiv \textbf{set}\ o.inv = T$$

$$Tr\,(\textbf{unpack}\ o\ \textbf{from}\ T) \equiv \textbf{set}\ o.inv = S$$

If Inv_T is the invariant of a class T, then

$$Tr(Inv_T) \equiv (inv \leq T) \rightarrow Inv_T$$

For all the other program constructs, Tr is the identity.

2.1.4 Translation's correctness

The following lemma allows one to omit the **assert** statements in the translation of the **pack** and **unpack** statements.

Lemma 1 *Let $\boldsymbol{P}$, S, σ, and σ' be a* Spec♯ *program, a program statement, and two program states, respectively. If $\boldsymbol{P} \vdash \sigma \xrightarrow{S} \sigma'$, then $Tr(\boldsymbol{P}) \vdash \sigma \xrightarrow{Tr(S)} \sigma'$.*

Proof. The proof runs by induction on the shape of the derivation tree for $\boldsymbol{P} \vdash \sigma \xrightarrow{S} \sigma'$. □

Theorem 1 *Let* $\boldsymbol{P}$ *be a* Spec$^\sharp{}_{\mathcal{S}}$ *program. Assume that* $\boldsymbol{P}$ *has legal assignments, admissible invariants, and legitimate* Spec♯ *executions. Then, Tr(*$\boldsymbol{P}$*) is a JML program with legal assignments, admissible invariants, and legitimate executions.*

Proof. $Tr(\mathbf{P})$ has legal assignments since every Spec♯ legal assignment is, according to Definition 5, a JML legal assignment. Moreover, the invariants of $Tr(\mathbf{P})$ are admissible since the corresponding invariants in Spec$^\sharp{}_{\mathcal{S}}$ are admissible and the *inv* fields used in the invariants are declared by the enclosing classes.

We now want to show that every execution of $Tr(\mathbf{P})$ is legitimate. To do that, we prove that for every object o and for every type T such that o is of type T, the following property holds in every state (not necessarily visible) of $Tr(\mathbf{P})$:

$$(o.inv \leq T) \rightarrow Inv_T(o) \tag{1}$$

The proof runs by induction over the sequence of program states.

Object creation: upon creating an object o, $o.inv =$ `Object`. This implies that (1) is preserved as `Object`'s invariant defaults to *true*.

Field assignment: Let $o'.f = exp$ be an assignment. Let us assume that this assignment affects an invariant $Inv_T(o)$, where o is of type T. Such an assignment can affect $Inv_T(o)$ if and only if $o' = o$ and Inv_T refers to `this`$.f$. According to Definition 2, the above assignment has the following precondition in Spec$^\sharp{}_{\mathcal{S}}$: $T' < o'.inv$, where T' is the class which declares f. As f is a field of o and o is of type T, we have $T \leq T'$ (otherwise $o.f$ cannot be type-checked). By this and $T' < o'.inv$, we get $T < o.inv$. This, however, implies that the invariant of class T for o, *i.e.*, $(o.inv \leq T) \rightarrow Inv_T(o)$, holds (since the right side of the implication evaluates to *false*). So, the property (1) is preserved.

The **set** *statement*:

- **set** statement resulted upon translating a Spec♯ **pack** statement: **pack** o **as** T.

  Such a **set** statement changes only the *inv* field of o. As Inv_T is not allowed to refer to any *inv* fields, the value of $Inv_T(o)$ cannot be changed by this **set** statement. On the other hand, $Inv_T(o)$ holds in the current execution of $Tr(\mathbf{P})$ since, by Lemma 1, it holds in the corresponding execution of $\mathbf{P}$. As the value of $Inv_T(o)$ cannot be changed, it means that $Inv_T(o)$ is *true* also after the **set** statement, and consequently, $(o.inv \leq T) \rightarrow Inv_T(o)$ holds. So, (1) is preserved.

- **set** statement resulted upon translating an Spec♯ **unpack** statement: **unpack** o **from** T.

  Similarly as in the above case, the **set** statement changes only the *inv* field of o. As discussed above, the value of $Inv_T(o)$ cannot be changed. On the other hand, the value of $o.inv$ is changed to the direct superclass of the class pointed to by $o.inv$ before the **set** statement. By Lemma 1, $o.inv$ is T in the current execution of $Tr(\mathbf{P})$ as that is case in the corresponding execution of $\mathbf{P}$. Consequently, the truth value of $o.inv \leq T$ might change, but only from *true* to *false*. So, $(o.inv \leq T) \rightarrow Inv_T(o)$ is true. Hence, (1) is preserved.

□

2.2 From JML to Spec♯

Task Given a JML program whose executions are legitimate in JML, we want to translate it (without changing its behavior) to a Spec♯ program whose executions are legitimate in Spec♯. In particular, we translate a source program that *implicitly* uses visible state semantics to a program that checks and uses visible state semantics *explicitly*.

2.2.1 Considered JML subset

The only difference with respect to the JML subset considered in Section 2.1.2 is concerning legal assignments:

Definition 9 (Legal assignment) *An assignment* $o.f = exp$ *is* legal *iff* $o = \texttt{this}$.

2.2.2 Considered Spec♯ subset

The considered subset is the same as the one defined in Section 2.1.1.

2.2.3 Translation

Definition 10 (Fully Pack and Unpack) *For a type* T *and an object* o *whose allocated type,* $type(o)$*, is a subclass of the classes* $T_1, \ldots, T_n$ *and* `Object`*, where* $type(o) <: T_1 <: \ldots <: T_n <:$ `Object`*, the statements* **fullyunpack** o *and* **fullypack** o **at** T *are defined as follows:*

fullyunpack o $\equiv$
  unpack o **from** $type(o)$
  unpack o **from** T_1
  $\vdots$
  unpack o **from** T_n
	fullypack o **at** $T \equiv$
	  pack o **as** T_n
	  pack o **as** T_{n-1}
	  $\vdots$
	  pack o **as** T
	  $o.inv := type(o)$

Definition 11 (Translation) *A method* m *declared in class* T *is translated as follows:*

Tr (
  $m(S\ x)$ {
    $Body$
  }
)
	$\equiv$	$m(S\ x)$
		  requires $\forall\ o\ \bullet\ o.inv = type(o)$
		  ensures $\forall\ o\ \bullet\ o.inv = type(o)$
		  {
		    fullyunpack `this`
		    $Tr\ (Body)$
		    fullypack `this` **at** T
		  }
		}

The declaration of the constructor in class T *is translated as follows:*

$$
Tr\left(\begin{array}{l} T(S\ x)\ \{ \\ \quad Body \\ \} \end{array}\right) \equiv \begin{array}{l} T(S\ x) \\ \quad \textbf{requires}\ \forall o \mid o \neq \texttt{this} \bullet o.inv = type(o) \\ \quad \textbf{ensures}\ \texttt{this}.inv = T \wedge \forall o \mid o \neq \texttt{this} \bullet o.inv = type(o) \\ \quad \{ \\ \qquad Tr\ (Body) \\ \qquad \textbf{pack}\ \texttt{this}\ \textbf{as}\ T \\ \quad \} \end{array}
$$

A method call $o.m()$ in a class T is translated as follows:

$$
Tr\left(\begin{array}{l} o.m(x) \end{array}\right) \equiv \begin{array}{l} \textbf{fullypack}\ \texttt{this}\ \textbf{at}\ T \\ o.m(x) \\ \textbf{fullyunpack}\ \texttt{this} \end{array}
$$

2.2.4 Discussion

Constructor's postcondition The constructors are required to build consistent objects. Therefore, the constructor of a class T should have the postcondition $inv = T$. To guarantee this postcondition, a **pack** statement is inserted just before the constructor returns.

Legal assignments A JML legal assignment is any assignment to `this`, whereas a Spec♯ legal assignment is every assignment to sufficiently unpacked objects. Given these definitions, one has to *sufficiently* unpack the JML's `this` receiver (before assigning to it) as opposed to simply unpack it. If `this`.inv is T upon entering a T's method and one simply unpack `this` (before assigning to it), then every assignment to `this`.f in T, with f declared by a T's superclass, would be illegal in Spec♯.

Let m be a JML method declared by a type T. Method m can assign to fields of `this` declared in different classes (T or T's superclasses). Therefore, `this` should be unpacked sufficiently enough to make legal all the assignments to `this`. For that, `this.inv` could just be set to the first class (in the type hierarchy) above the highest class which declares a field m assigns to. To simplify the exposition, we set `this.inv` to the topmost type of the type hierarchy, *i.e.*, `Object`.

Virtual methods Without introducing the statements **fullypack** and **fullyunpack**, the precondition $inv = 1$ for virtual methods, proposed as *default* by Barnett *et al.* [2], would not always work. Let us consider the Spec♯ example in Figure 1.

Upon entering $Super :: foo$, one can assume that `this`.inv is $Super$. For the virtual call to bar, $Super :: foo$ needs to ensure $this.inv = type($`this`$)$. Obviously, this precondition does not hold in our example (it would hold if and only if class $Super$ was **sealed**). Note that a simple **pack**-ing of `this` upon entering $Super :: foo$ does not help since $type($`this`$)$ is not necessarily Sub: for example, `this`.inv could be a proper subclass of Sub that implements $Super :: foo$ by calling the implementation of foo in Sub. As $type($`this`$)$ is not known at the time of the virtual call $bar()$, the number of **pack** statements needed to modify `this`.inv from $Super$ to $type($`this`$)$ is not known either. However, this brings up again the necessity of introducing the **fullypack** statement.

If $Super :: foo$ updates inherited fields before the call to bar, then `this` would have to be sufficiently unpacked at the beginning of $Super :: foo$. As the updated fields can be declared

Figure 1 Virtual methods with precondition $inv = 1$.

```
class Super {
    virtual foo()
        requires inv = 1 {
            bar();
        }

    virtual bar()
        requires inv = 1 {}
}
```

```
class Sub : Super {
    foo()
        requires inv = Sub {
            unpack this from Sub
            base.foo();
            pack this as Sub
        }

    bar()
        requires inv = Sub {}
}
```

by any of *Super*'s superclasses, `this` would have to be fully unpacked. Thus, a **fullyunpack** statement would have to be inserted.

In the context of virtual methods having the precondition $inv = 1$, supercalls become problematic. When an implementation of a virtual method is invoked through a supercall, the `this` object is not in a consistent state. That means that although all the invariants for `this` hold in the JML source program, the Spec♯ methodology cannot ensure the same property for the corresponding Spec♯ program. This downside could, however, be fixed through our translation, namely by inserting before every supercall an **assert** statement that checks the invariant of the enclosing class. In our example, as `this`.*inv* is *Super* at the time of the call **base**.*foo*(), the statement **assert** Inv_{Sub}(`this`) could be inserted just before the supercall.

Hence, setting the precondition $inv = 1$ for virtual methods would still require the statements **fullypack** and **fullyunpack**. As such a precondition would also necessitate the **assert**s before several supercalls, we have decided to set the precondition $\forall o \bullet o.inv = type(o)$ also for virtual methods.

Modular verification The statement **fullypack** o *cannot* be defined exactly in the same way as the statement **fullyunpack** o, that is as a sequence of statements **pack** o **as** T_i, $i = n..1$. Such a definition would be problematic for modular verification. Such a **fullypack** statement, invoked, for example, with the `this` object in a method m of a class T, has to check, in particular, the invariants of all T's strict subclasses from T to *type*(`this`). Checking these invariants requires possessing information about them. This yields, however, a problem in modular verification. This downside does not, however, show up in the Spec♯ programs generated by our translation. That is because, according to Definition 6, the JML admissible invariants are only allowed to depend on fields declared by the enclosing class. This means that, in the resulting Spec♯ programs, the inherited fields are never referred to in invariants. In the Spec♯ methodology, this would be equivalent with assuming that every field is non-*additive*. So, the method m cannot modify the invariants of T's strict subclasses. Therefore, the **fullypack** statement does not need to check these invariants. Consequently, a statement **fullypack** o **at** T comprises only **pack** statements from T_n to T (see Definition 10). Note that **fullypack** o sets `this`.*inv* directly as opposed to set it via **pack** statements.

Translation's naturalness Although the translation scheme requires the introduction of the statements **fullypack** and **fullyunpack**, we think that the translation is *natural* in the sense that the Spec♯ programs generated by the translation are verified through the Spec♯ methodology along the lines the JML source program is verified with the JML methodology. Thus, the JML `this` object is fully exposed between visible states, so is the corresponding Spec♯ `this` object as a result of applying **fullyunpack** and **fullypack**. Moreover, the invariants of all the enclosing class' superclasses are checked for the JML `this` object when passing to a new visible state, so are the corresponding invariants for the Spec♯ `this` object upon invoking **fullypack**. In this sense, the method precondition $\forall\ o\ \bullet\ o.inv = type(o)$ prescribed by the translation *explicitly* expresses the *implicit* JML visible state semantics.

2.2.5 Translation's correctness

Lemma 2 *Let m be a method in a JML program. In $Tr(m)$,* `this`*.inv is* `Object` *before and after the execution of every m's statement other than a method call.*

Proof. By induction on number of the visible states of the execution. □

Theorem 2 *Let $\boldsymbol{P}$ be a JML program. If $\boldsymbol{P}$ has legal assignments, admissible invariants and legitimate JML executions, then $Tr(\boldsymbol{P})$ is a* Spec♯ *program with legal assignments, admissible invariants, and legitimate executions.*

Proof. Based on Tr's definition for method declarations (Definition 11), every legal JML assignment in $\mathbf{P}$ is, in particular, a legal Spec♯ assignment in $Tr(\mathbf{P})$. This is because in $Tr(\mathbf{P})$, every assignment occurs between a **fullyunpack** and a **fullypack**. So, by Lemma 2, it occurs when `this`.inv is `Object`. Consequently, an arbitrary assignment to `this`.f is legal according to Definition 2 since $T <$ `Object`, where T is the class which declares the field f.

Every JML admissible invariant (Definition 6) is, in particular, a Spec♯ admissible invariant (Definition 3).

To conclude the proof, we show that the following two properties hold for an arbitrary execution of $Tr(\mathbf{P})$:

(P1) the execution is legitimate in Spec♯ according to Definition 4, *i.e.*, the **assert**s prescribed by the **unpack** and **pack** statements do not fail at run-time;

(P2) the precondition and postcondition of every method called in the given execution of $Tr(\mathbf{P})$ are ensured;

These properties can be simultaneously proved by induction on the executions' length, *i.e.*, the number of visible states of these executions.

The base case: the execution has a single state, *i.e.*, the initial state. This state can only be the prestate of a constructor's execution. As this state does not involve any **unpack** or **pack** statements, (P1) obviously holds. Also, (P2) is satisfied since the constructor has no precondition.

The induction step: we assume that the properties (P1) and (P2) are satisfied in executions of length strictly less than n. Without losing generality, one may assume that there is a method m such that the nth visible state is either m's poststate or the prestate of a method, say m', called by m. As the proofs of both cases are similar, we only treat here the second one. Let T be the class of m. According to our translation scheme, between the (n-1)th state and the nth state, the only executed **pack** and **unpack** statements are those within the **fullypack**

operation inserted just before m calls m' and the **fullyunpack** operation executed when m' starts executing, respectively.

The **assert**s imposed by the **pack** operations within **fullypack** `this`, where `this` is the receiver object of m, do not fail at run-time since

- by Lemma 2, `this`.inv is `Object` before **fullypack** `this`, and
- the invariants of the (not necessarily proper) superclasses of T checked by the **pack** operations hold by the visible state semantics of the JML source program (Definition 7).

Moreover, the precondition of m', *i.e.*, $\forall\, o \bullet o.inv = type(o)$, is ensured by m since

- the `this` object of m is the only object whose inv field can be modified between the (n-1)th state and the nth state, and
- `this`.inv is updated to $type$(`this`) by the statement **fullypack** `this`.

The **assert**s prescribed by the **unpack** operations within **fullyunpack** `this`, where `this` is the receiver object of m', do not fail at run-time since, by the precondition of m', `this`.inv is $type$(`this`) before **fullyunpack** `this`.

We also need to prove the following properties:

- the **assert**s prescribed by the **pack** at the end of every constructor succeeds;
- the postcondition of every constructor is guaranteed;

The second property follows immediately from the first one.

Let us consider the constructor of a class T. The **assert** checking the value of `this`.inv succeeds since `this`.$inv = S$, where S is the direct superclass of T. This can be easily proved by induction over the "depth" of S in the subtyping hierarchy, where the following fact is used: in JML, it is ensured that every constructor (implicitly or explicitly) calls its superclass constructor. Moreover, the **assert** testing the invariant of T does not fail since Inv_T(`this`) holds by the visible state semantics of the JML source program (Definition 7). □

3 Object structures

3.1 From Spec♯ to JML

3.1.1 Considered Spec♯ subset

For an object p, $p.owner.obj$ and $p.owner.typ$ denote p's owning object and the class of the owning object that induces the ownership, respectively. More exactly, if $p.owner = [o, T]$, then $p.owner.obj = o$ and $p.owner.typ = T$.

The methodology described in [2, 4] uses a special boolean field, $committed$, to indicate whether the object is committed. According to this methodology, if $p.owner = [o, T]$, then $p.committed$ if and only if $o.inv \leq T$. By using the components obj and typ, this definition becomes as follows:

$$p.committed :\Longleftrightarrow p.owner.obj.inv \leq p.owner.typ$$

As the field *committed* can be expressed in terms of *inv* and *owner*, the methodology we present here omits it.

[Constructors] Every constructor has the postcondition

$$inv = T \wedge owner.typ < owner.obj.inv$$

where T is the class of the constructor.

To ensure that every assignment from the considered Spec♯ subset is legal in JML, we only consider Spec♯ legal assignments that assign to fields of `this`.

Definition 12 (Legal assignment) *If f is declared in class T, then the assignment $o.f = exp$ is* legal *iff $T < o.inv$ and $o =$ `this`.*

Definition 13 *Let T and S be two classes such that S is the direct superclass of T. The statements* **pack** *and* **unpack** *are defined as follows:*

pack o **as** $T \equiv$
  assert $o \neq null \wedge o.inv = S$
  assert $Inv_T(o)$
  assert $(\forall p \mid p.owner = [o, T] \bullet p.inv = type(p))$
  $o.inv := T$

unpack o **from** $T \equiv$
  assert $o \neq null \wedge o.inv = T \wedge o.owner.typ < o.owner.obj.inv$
  $o.inv := S$

To ensure that the Spec♯ invariants are mapped into admissible JML invariants, the only accesses to `this`'s fields that we consider in the Spec♯ admissible expressions are those for which the field is declared by the enclosing class.

Definition 14 (Ownership admissible invariant) *An invariant in a class T is* ownership admissible *iff each of its access expressions is of one of the following forms:*

- `this`*.f, where f is a field declared by T;*
- `this`*.$g_0 \ldots g_n.f$, where g_0 is declared by T and $(g_i)_{i=0}^{n}$ are* `rep` *fields.*

Assumption The Spec♯ subset we consider here does not include the **transfer** statement. This is because the Spec♯ programs that perform ownership transfer cannot be translated to JML as the JML's ownership graphs are immutable.

Future work An extension of the JML's Universe Type System that supports ownership transfer has been proposed in [5]. Potential future work includes defining a translation scheme that maps the Spec♯ **transfer** statement in the JML set that supports ownership transfer.

3.1.2 Considered JML subset

Assumption To ensure *the subclass separation principle*, according [6], the `rep` fields should be declared **private**.

Future work Relaxing the above restriction on `rep` fields is part of future work. This could be achieved, for example, by defining owners as pairs consisting of an object reference and a class name.

Definition 15 (Legal assignment) *An assignment* $o.f = exp$ *is* legal *iff* $o =$ `this`.

The Universe Type System the JML methodology is based on enforces several typing constraints (see [6]). We assume all these constraints. In particular, we assume the following properties (Figure 6 and Corollary 5.2 in [6]):

[Legal field access] A field access $o.f$ appearing in a method m is legal iff one of the following conditions is satisfied:

- o's type is a `peer` type and f's type is not a `rep` type, or
- o is m's receiver and f's type is a `rep` type, or
- o's type is a `rep` type and f's type is not a `rep` type.

The type constraints imposed by the Universe Type System restrict the set of allowed method calls:

[Legal method call] A method m can only call methods on a `rep` or `peer` object of m's receiver object.

Definition 16 (Ownership admissible expression) *An access expression* `this`$.g_0 \ldots g_n$ *appearing in a class* T *is* ownership admissible *if* g_0 *is declared by* T *and*

- $n = 0$*, or*
- $n > 0$ *and* g_0 *is declared as a* `rep` *field.*

Definition 17 (Ownership admissible invariant) *An invariant in a class* T *is* ownership admissible *iff each of its access expressions is ownership admissible.*

Definition 18 (Relevant object) *An object* o *is* relevant *to the execution of a method* m *iff* o *is inside the context in which* m *executes.*

Definition 19 (Legitimate JML program execution) *A program execution is* legitimate *iff for each execution of a method* m *and for each object* o *relevant to* m*'s execution,* o *satisfies its invariant in the prestate and poststate of* m*'s execution.*

3.1.3 Translation

[**Idea**] Consider in class `Object` the ghost fields *inv* and *owner* of declared type `Type` and `Object`, respectively.

We assume that every Spec$\sharp$ `rep` field is declared **private**. Moreover, to ensure that the translated Spec$\sharp$ expressions are legal in JML, we impose that they are typeable in the JML's Universe Type System.

Definition 20 (Translation) *The* **pack** *and* **unpack** *statements are translated as follows:*

$$Tr\,(\textbf{pack}\ o\ \textbf{as}\ T) \equiv \textbf{set}\ o.inv = T$$

$$Tr\,(\textbf{unpack}\ o\ \textbf{from}\ T) \equiv \textbf{set}\ o.inv = S$$

If Inv_T is the invariant of a class T, then

$$Tr(Inv_T) \equiv (inv \leq T) \rightarrow (Inv_T \wedge \forall o' \mid o'.owner = [\texttt{this}, T] \bullet o'.inv = type(o'))$$

The translation of a Spec$\sharp$ invariant essentially represents the inlining of the program invariants defined in [4].

3.1.4 Translation's correctness

Theorem 3 *Let* $\boldsymbol{P}$ *be a* Spec$\sharp$ *program. Assume that* $\boldsymbol{P}$ *has legal assignments, admissible invariants, and legitimate* Spec$\sharp$ *executions. Then,* $Tr(\boldsymbol{P})$ *is a JML program that is type correct in the Universe Type System and has legal assignments, admissible invariants, and legitimate executions.*

Proof. Every assignment in $Tr(\mathbf{P})$ is legal as we have considered Spec$\sharp$ programs which assign to `this` only (Definition 12). Moreover, $Tr(\mathbf{P})$ has admissible invariants since every Spec$\sharp$ ownership admissible expression is, in particular, a JML ownership admissible expression.
Based on legitimateness of $\mathbf{P}$'s executions, we want to prove now that every execution of $Tr(\mathbf{P})$ is legitimate. For this, we show that for every object o and every type T such that o is of type T, the property

$$(o.inv \leq T) \rightarrow (Inv_T(o) \wedge \forall o' \mid o'.owner = [o, T] \bullet o'.inv = type(o')) \tag{2}$$

is satisfied in every state of $Tr(\mathbf{P})$, in particular in every visible state. The proof runs by induction over the sequence of program states.

Object creation: upon creating an object o, $o.inv =$ `Object`. This implies that (2) is preserved since `Object`'s invariant defaults to *true* and there is no object o owns.

Field assignment: Let $o'.f = exp$ be a (legal) assignment, where field f is declared by a class F. Let us consider the effect of this assignment on an invariant $Inv_T(o)$ for some o of type T (note that the formula $\forall o' \mid o'.owner = [o, T] \bullet o'.inv = type(o')$ cannot be affected by our field assignment). We would like to show that if $Inv_T(o)$ refers to the object pointed by $o'.f$, then o is sufficiently unpacked, *i.e.*, $T < o.inv$. Following the definition of Spec♯ admissible expressions, we analyze the following cases:

1. Inv_T refers to `this`.f and $o' = o$. As the assignment $o'.f = exp$ is legal in Spec♯, $F < o'.inv = o.inv$. Class T is a subclass of F since the expression $o'.f$ should typecheck. Consequently, $T < o.inv$.
2. Inv_T refers to `this`.$g_0 \ldots g_n.f$, where g_i, $i = 0..n$ are `rep` fields, and $o.g_0 \ldots g_n = o'$. As the Spec♯ assignment to o' is legal, $F < o'.inv$. So, o' is not consistent. By the definition of **unpack**, $o.g_0 \ldots g_{n-1}$ is not consistent either. This argument can be inductively applied to derive that $o.g_0$ is not consistent. By the **unpack**'s definition, the owner of $o.g_0$, *i.e.*, o, should be unpacked beyond the owner type, *i.e.*, the type of the `rep` field g_0. That means $S < o.inv$, where S is the class that declares g_0. For $o.g_0$ to typecheck, S should be a (not necessarily proper) subclass of T: $T \leq S$. Consequently, $T < o.inv$.

The **set** *statement*:

- **set** statement resulted upon translating a Spec♯ **pack** statement: **pack** o **as** T.

  This **set** statement can only change the *inv* field of o. As the *inv* fields are not allowed to appear in invariants, the value of $Inv_T(o)$ cannot be changed. Note that in the corresponding **P**'s execution, $Inv_T(o)$ holds since the execution is legitimate and $Inv_T(o)$ is ensured through an **assert**. By Lemma 1, $Inv_T(o)$ shall also hold in the given execution of $Tr(\mathbf{P})$.

  We also have to show that (2) holds for the owner of o. If o' and S are such that $o.owner$ = [o',S], then

  $$(o'.inv \leq S) \rightarrow (Inv_S(o') \wedge \forall o'' \mid o''.owner = [o'', S] \bullet o''.inv = type(o''))$$

  holds even if $o.inv$ has been modified. The above formula holds since $S < o'.inv$. This is because, according to the definition of the **unpack** statements (our **pack** statement is necessarily preceded by an **unpack** statement), the owner object shall be unpacked beyond the owning type: in our case, $o.owner.typ < o.owner.obj.inv$.

- **set** statement obtained upon translating a Spec♯ **unpack** statement: **unpack** o **from** T.

  Like in the above case, the **set** statement can only change o's *inv* field. As discussed above, the truth value of $Inv_T(o)$ cannot be changed. By Lemma 1, $o.inv$ is T before the **set** statement. This statement updates $o.inv$ to T's direct superclass. This means that the truth value of $o.inv$ is changed from *true* to *false*. So, (2) is preserved.

  Similarly as in the case of the **pack** statement, (2) can be proved to hold for o's owner as the owner is necessarily unpacked beyond the owning type of o.

This concludes the proof.

□

3.2 From JML to Spec♯

3.2.1 Considered Spec♯ subset

The only difference with respect to the Spec♯ subset considered in Section 3.1.1 is concerning the ownership admissible invariant:

Definition 21 (Ownership admissible invariant) *An invariant in a class T is* ownership admissible *iff each of its access expressions is of one of the following forms:*

- $\texttt{this}.f$*, where f is a field declared by T or T's superclasses;*
- $\texttt{this}.g_0 \ldots g_n.f$*, where $(g_i)_{i=0}^{n}$ are* `rep` *fields.*

3.2.2 Considered JML subset

The considered subset is the same as the one defined in Section 3.1.2.

3.2.3 Translation

Definition 22 (Peers) *For two objects o and o', we define $arePeers(o, o')$ as follows:*

$$arePeers(o, o') = \exists q, T, T' \bullet o.owner = [q, T] \wedge o'.owner = [q, T']$$

Definition 23 (Translation) *A type T which declares the* `peer` *fields $g_0, \ldots, g_n$ is translated as follows:*

$$
\begin{array}{lcl}
\begin{array}{l}
Tr\,(\\
\quad T\,\{\\
\quad\quad \textbf{peer}\ T_0\ g_0;\\
\quad\quad \vdots\\
\quad\quad \textbf{peer}\ T_n\ g_n;\\
\quad\quad \vdots\\
\quad\quad RoutineDeclarations\\
\quad \}\\
)
\end{array}
& \equiv &
\begin{array}{l}
T\,\{\\
\quad T_0\ g_0;\\
\quad \vdots\\
\quad T_n\ g_n;\\
\quad \textbf{invariant}\ \forall i = 0, n \bullet \texttt{this}.g_i.owner = \texttt{this}.owner\\
\quad \vdots\\
\quad Tr(RoutineDeclarations)\\
\}
\end{array}
\end{array}
$$

The declaration of a method m in class T is translated as follows:

$$
\begin{array}{lcl}
\begin{array}{l}
Tr\,(\\
\quad m(S\ x)\,\{\\
\quad\quad Body\\
\quad \}\\
)
\end{array}
& \equiv &
\begin{array}{l}
m(S\ x)\\
\quad \textbf{requires}\ (\texttt{this}.owner.typ < \texttt{this}.owner.obj.inv) \wedge\\
\quad\quad\quad\quad \forall o \mid arePeers(o, \texttt{this}) \bullet o.inv = type(o)\\
\quad \textbf{ensures}\ (\texttt{this}.owner.typ < \texttt{this}.owner.obj.inv) \wedge\\
\quad\quad\quad\quad \forall o \mid arePeers(o, \texttt{this}) \bullet o.inv = type(o)\\
\quad \{\\
\quad\quad \textbf{fullyunpack}\ \texttt{this}\\
\quad\quad Tr\,(Body)\\
\quad\quad \textbf{fullypack}\ \texttt{this}\ \textbf{at}\ T\\
\quad \}
\end{array}
\end{array}
$$

The declaration of the constructor in class T is translated as follows:

$$
\begin{array}{lcl}
\begin{array}{l}
\mathit{Tr}\ (\\
\quad T(S\ x)\ \{ \\
\quad\quad \mathit{Body} \\
\quad \} \\
)
\end{array}
& \equiv &
\begin{array}{l}
T(S\ x) \\
\quad \textbf{requires}\ \forall o \mid o \neq \texttt{this} \wedge \mathit{arePeers}(\texttt{this}, o) \bullet o.\mathit{inv} = \mathit{type}(o) \\
\quad \textbf{ensures}\ (\texttt{this}.\mathit{owner}.\mathit{typ} < \texttt{this}.\mathit{owner}.\mathit{obj}.\mathit{inv}) \wedge \\
\quad\quad\quad\quad \texttt{this}.\mathit{inv} = T \wedge \\
\quad\quad\quad\quad \forall o \mid o \neq \texttt{this} \wedge \mathit{arePeers}(\texttt{this}, o) \bullet o.\mathit{inv} = \mathit{type}(o) \\
\quad \{ \\
\quad\quad \mathit{Tr}\ (\mathit{Body}) \\
\quad\quad \textbf{pack}\ \texttt{this}\ \textbf{as}\ T \\
\quad \}
\end{array}
\end{array}
$$

A method call $o.m(x)$ appearing in class T is translated as follows:

$$
\begin{array}{lcl}
\begin{array}{l}
\mathit{Tr}\ (\\
\quad o.m(x) \\
)
\end{array}
& \equiv &
\begin{array}{l}
\textbf{if}\ \mathit{arePeers}(o, \texttt{this})\ \textbf{then} \\
\quad \textbf{fullypack}\ \texttt{this}\ \textbf{at}\ T \\
o.m(x) \\
\textbf{if}\ \mathit{arePeers}(o, \texttt{this})\ \textbf{then} \\
\quad \textbf{fullyunpack}\ \texttt{this}
\end{array}
\end{array}
$$

3.2.4 Translation's correctness

The following lemma is an immediate consequence of Definition 22:

Lemma 3 *If $arePeers(o, o')$ and $arePeers(o', o'')$, then $arePeers(o, o'')$.*

The property claimed by the following lemma is a consequence of the Spec♯ methodology (more exactly, of the definitions of **pack** and **unpack**) and not of translation *Tr*.

Lemma 4 *Given a JML program $\boldsymbol{P}$, the following property is a program invariant in $Tr(\boldsymbol{P})$:*

$$\forall o, o' \bullet [arePeers(o, o') \rightarrow [(o.owner.typ < o.owner.obj.inv) \leftrightarrow (o'.owner.typ < o'.owner.obj.inv)]]$$

Proof. By $arePeers(o, o')$, we get

$$o.owner.obj = o'.owner.obj$$

The proof runs by induction on the execution's length. □

The definition of the translation scheme *Tr* enforces the following lemma.

Lemma 5 *Let m be a method in a JML program. When executing $Tr(m)$, the following properties hold before and after every m's statement that is not a method call:*

(1) $\texttt{this}.owner.typ < \texttt{this}.owner.obj.inv$

(2) $\texttt{this}.inv = \texttt{Object}$

(3) $\forall o \bullet [\exists T \bullet (o.owner = [\texttt{this}, T]) \rightarrow (o.inv = type(o))]$

(4) $\forall o \bullet [(o \neq \texttt{this} \wedge arePeers(o, \texttt{this})) \rightarrow (o.inv = type(o))]$

Proof. The proof runs by induction on the execution's length. □

Theorem 4 *Let **P** be a JML program that is type correct in the Universe Type System. If **P** has legal assignments, admissible invariants and legitimate JML executions, then $Tr(\boldsymbol{P})$ is a* Spec♯ *program with legal assignments, admissible invariants, and legitimate executions.*

Proof. Every JML legal assignment in **P** is, in particular, a Spec♯ legal assignment in $Tr(\mathbf{P})$. This is since in $Tr(\mathbf{P})$, every assignment to `this` occurs between a **fullyunpack** `this` and a **fullypack** `this` statement. So, by property (2) in Lemma 5, it occurs when `this`.inv is `Object`. Consequently, an arbitrary assignment to `this`.f is legal according to Definition 12 since $T <$ `Object`, where T is the class which declares the field f. Note that $T \neq$ `Object` as `Object` does not declare any fields.

Every JML admissible invariant (Definition 17) is, in particular, a Spec♯ admissible invariant (Definition 14). This is since every access expression permitted in a JML admissible invariant is also allowed in a Spec♯ admissible invariant. To show that, let us consider the invariant of a JML type T:

- if `this`.g_0 occurs in the JML invariant, where g_0 is declared by T, then `this`.g_0 is also allowed to appear in the invariant of the corresponding Spec♯ type.
- if the expression `this`.$g_0 \ldots g_n$ appears in the JML invariant, where g_0 is declared as `rep`, then `this`.$g_0 \ldots g_n$ is also allowed in the invariant of the corresponding Spec♯ type.

We now prove the following two properties for an arbitrary execution of $Tr(\mathbf{P})$:

(P1) the execution is legitimate in Spec♯, *i.e.*, the **assert**s prescribed by the **unpack** and **pack** statements do not fail at run-time;

(P2) the precondition and postcondition of every method called in the given execution of $Tr(\mathbf{P})$ are ensured;

(P1) and (P2) can be simultaneously proved by induction on the execution's length.

The base case: the execution has a single state, the initial state. This state can only be the prestate of a constructor's execution. As this state does not involve any **unpack** or **pack** statements, (P1) obviously holds. Also, (P2) holds as the constructor's precondition is satisfied. This is so since there is no allocated object in this state.

The induction step: we assume that the properties (P1) and (P2) are satisfied in executions of length strictly less than n. Without losing generality, one assumes there is a method m such that the nth visible state is either m's poststate or the prestate of a method m' called by m. Let T be the class of m.

Case 1: the nth visible state is the prestate of the call to m'. The call of m' is legal in JML iff the object m' is called on, say o, is either a `rep` or a `peer` of m's receiver object.

Case 1.1: We first assume that o is a `rep` of m's `this` object: so, there exists T' such that $o.owner = [\texttt{this}, T']$.
The precondition of m' is ensured by m since

- $o.owner.typ < o.owner.obj.inv$: this is equivalent with $T' <$ `this`$.inv$ which is a consequence of property (2) in Lemma 5 (note that T' cannot be Object).

- $o.inv = type(o)$: this is a consequence of property (4) in Lemma 5 which claims that o is consistent before the call of m'.

- $o'.inv = type(o')$ for every $o' \neq o$ such that $arePeers(o, o')$. In other words, every `peer` object of o is consistent. This can be shown as follows. Let us consider an arbitrary `peer` o' of o. By Definition 22, there exists S, S' and q such that $o.owner = [q, S]$ and $o'.owner = [q, S']$. On the other hand, $o.owner = [$`this`$, T']$. It follows that q is `this`, and consequently o' is a `peer` of `this`. By property (4) in Lemma 5, we get that o' is consistent.

Hence, property (P2) is preserved.

Between the (n-1)th visible state and nth visible state, there no **pack** statements and the only **unpack** statements are those within the operation **fullyunpack** o executed upon the entry of m'. The **assert**s prescribed by this **fullyunpack** do not fail at run-time since, by the precondition of m' (proved to hold above), $o.inv = type(o)$ and $o.owner.typ < o.owner.obj.inv$. So, property (P1) is maintained.

Case 1.2: We now assume that o is a `peer` of m's `this` object: so, there exists o', T', T'' such that `this`$.owner = [o', T']$ and $o.owner = [o', T'']$.

According to the translation scheme, the only **pack** and **unpack** statements executed between the (n-1)th state and nth state are those prescribed by **fullypack** `this` **at** T executed before the call of m', where `this` is m's receiver object, and **fullyunpack** o executed upon the entry of m'.

The **assert**s imposed by the **pack** operations within **fullypack** `this` **at** T do not fail at run-time since

- `this`$.inv =$ `Object`: this follows from property (2) in Lemma 5.

- every object owned by `this` is consistent: this follows from property (4) in Lemma 5.

- the invariant $Inv_S($`this`$)$ holds for every type S that is a supertype of T. This is a consequence of the relevant invariant semantics of the JML program **P**. Note that, by the subclass separation principle, also the invariants of the subclasses between T and $type(o)$ hold. However, to allow modular reasoning, these invariants are not checked (through **assert** statements).

Moreover, the precondition of m' is guaranteed by m since

- $o.owner.typ < o.owner.obj.inv$: this is a consequence of Lemma 4 and property (1) in Lemma 5.

- $o.inv = type(o)$: this follows from property (4) in Lemma 5.

- $o'.inv = type(o')$ for every $o' \neq o$ such that $arePeers(o, o')$. In other words, every `peer` of o is consistent. This can be derived from property (4) in Lemma 5 and from the fact that `this` becomes consistent after **fullypack** `this` is executed.

The **assert**s prescribed by the **unpack** operations within **fullyunpack** o do not fail at run-time since, by the precondition of m', $o.owner.typ < o.owner.obj.inv$ and $o.inv$ is $type(o)$ (before **fullyunpack** `this`).

So, properties (P1) and (P2) are preserved.

Case 2: the nth visible state is the poststate of m. Let m'' be the method that called m and o be the receiver object of m''. The call of m is legal in JML iff the `this` object of m is either a `rep` or a `peer` of o.

Case 2.1: We first assume that `this` is a `rep` of o: so, there exists T' such that $o.owner = [$`this`$, T']$.

According to the translation scheme, the only **pack** and **unpack** statements executed between the $(n$-1)th state and nth state are those prescribed by **fullypack** `this` **at** T executed upon exiting m.

The **assert**s imposed by the **pack** operations within the above **fullypack** operation do not fail at run-time since

- `this`$.inv =$ `Object`: this follows from property (2) in Lemma 5.
- every object owned by `this` is consistent: this follows from property (3) in Lemma 5.
- the invariant $Inv_S($`this`$)$ hold for every type S that is a supertype of T. This is a consequence of the relevant invariant semantics of the JML program **P**. Similarly as in *Case 1.2*, the invariants of T's subclasses are not checked explicitly.

So, (P1) is preserved.

The postcondition of m is ensured since

- `this`$.owner.typ <$ `this`$.owner.obj.inv$: this follows from property (1) in Lemma 5.
- `this`$.inv = type($`this`$)$: this obviously holds after the execution of **fullypack** `this` **at** T at the end of m.
- $o'.inv = type(o')$ for every $o' \neq$ `this` such that $arePeers($`this`$, o')$. This can be obtained from property (4) in Lemma 5.

So, property (P2) is maintained.

Case 2.2: We assume that `this` is a `peer` of o. One can prove as above that (P2) is preserved. According to the translation scheme, the only **pack** and **unpack** statements executed between the $(n$-1)th state and nth state are those prescribed by **fullypack** `this` **at** T executed upon exiting m and **fullyunpack** o executed upon transferring control back to m''. Similarly as in *Case 2.1*, one can prove that the **assert**s implied by the **fullypack** operation succeed at run-time.

The **assert**s of **fullyunpack** o do not fail at run-time since

- $o.owner.typ < o.owner.obj.inv$: this follows from Lemma 4 and property (1) in Lemma 5.
- $o.inv = type(o)$: this follows from the postcondition of m since o and the `this` object of m are `peer`s.

So, the property (P1) is preserved.

This concludes the proof. $\square$

4 Visibility-based invariants

4.1 From Spec♯ to JML

4.1.1 Considered Spec♯ subset

In addition to the Spec♯ `rep` field annotations considered in Section 3, Spec♯ includes explicitly `peer` field annotations to express that the object stored in the `peer` field and the receiver object have the same owner.

Definition 24 (Visibility admissible expression) *An access expression appearing in a class T is* visibility admissible *if it is of the form* `this`$.g_1 \dots g_n.f$*, where*

(1) $n = 0$*, or*

(2) g_1 *is a* `rep` *field and each of the fields* g_i*,* $i = 2, n$ *is a* `rep` *or a* `peer` *field.*

The field f must not be the predefined field inv.

Definition 25 (Visibility admissible invariants) *An invariant in a class T is* visibility admissible *if each of its access expressions is visibility admissible.*

Definition 26 (Legal assignment) *If f is declared in class T, then the assignment $o.f = exp$ is* legal *iff $T < o.inv$.*

4.1.2 Considered JML subset

Definition 27 (Visibility admissible expression) *An access expression* `this`$.g_0 \dots g_n$ *appearing in a class T is* visibility admissible *if g_0 is declared by T and the access expression is of one of the following forms:*

(1) $n = 0$*, or*

(2) every field g_i*,* $i = 0, n-1$*, is a* `peer` *field, or*

(3) $n > 0$ *and* g_0 *is declared as a* `rep` *field.*

Remark 1 *Unlike the corresponding definition in [6], Definition 27 includes also the case $n = 0$. That is to allow expressions of the form* `this`$.f$*, where f is a* `rep` *field. As a result, every visibility admissible expression is, in particular, an ownership admissible expression (see Definition 16).*

Definition 28 (Visibility admissible invariants) *An invariant in a class T is* visibility admissible *if each of its access expressions is visibility admissible and if, for each prefix of an access expression which appears in the invariant and matches form (1) in Definition 27, the invariant is visible in the class which declares the corresponding field g_n.*

To simplify the approach, we assume that every module imports all the other modules. That allows one to omit the *visibility* notion.

Definition 29 (Legal assignment) *An assignment $o.f = exp$ is* legal *iff o and the* `this` *object of the enclosing method are peers.*

4.1.3 Translation

The idea is the same as in Section 3.1.3, namely to consider in class `Object` the ghost fields *inv* and *owner* of declared type `Type` and `Object`, respectively. The translation *Tr* eliminates the `dependent` and *owner*-`dependent` clauses as they are not needed anymore in JML.

Similarly as in Section 3.1.3, we assume that every Spec♯ `rep` field is **private** and every Spec♯ expression is typeable in the JML's Universe Type System.

The translation scheme we are looking for should be defined such that the following theorem holds:

Theorem 5 *Let* $\boldsymbol{P}$ *be a* Spec♯ *program. Assume that* $\boldsymbol{P}$ *has legal assignments, admissible invariants, and legitimate* Spec♯ *executions. Then,* $Tr(\boldsymbol{P})$ *is a JML program that is type correct in the Universe Type System and has legal assignments, admissible invariants, and legitimate executions.*

Discussion Concerning the proof of Theorem 5, one can easily show that every admissible invariant of the Spec♯ program $\mathbf{P}$ (see Definition 25), is, in particular, an admissible invariant of the JML program $Tr(\mathbf{P})$ (see Definition 28). Moreover, the proof of $Tr(\mathbf{P})$ executions' legitimateness can be done in the same way as the corresponding proof part of Theorem 3.

However, *one open problem* is to define (possibly restrict or extend) the sets of legal assignments specified by Definitions 26 and 29 such that *Tr* maps the Spec♯ assignments considered legal to JML assignments considered legal. The proof obligations imposed by the legal assignments should, however, preserve the soundness of the JML and Spec♯ methodologies.

4.2 From JML to Spec♯

4.2.1 Considered JML subset

The JML set considered is the same as the one defined in Section 4.1.2.

4.2.2 Considered Spec♯ subset

Definition 30 (Visibility admissible expression) *An access expression appearing in a class* T *is* visibility admissible *if it has one of the following forms:*

(1) $\texttt{this}.g_1 \ldots g_n.f$*, where* $n = 0$ *or* g_1 *is a* `rep` *field and each of the fields* g_i*,* $i = 2, n$ *is a* `rep` *or a* `peer` *field.*

(2) $\texttt{this}.g_1 \ldots g_n.f$*, where* $n \geq 1$*,* f *is different than the field owner, and* T *appears in* f*'s* `dependent`*-clause.*

(3) $\texttt{this}.g_1 \ldots g_n.owner$*, where* $n \geq 1$ *and* T *is in an owner-*`dependent` *declaration of* g_n*'s type.*

(4) $x.f$*, where* x *is bound by a universal quantification of the form*

$$\forall T\, x \mid x.owner = [\texttt{this}.T'] \bullet P(x)$$

and T' *is a superclass of* T*.* $P(x)$ *may refer to the identity and the state of* x*, but not to the states of objects referenced by* x*.*

The field f *must not be the predefined field inv.*

The definition of visibility admissible invariants is then based on the definition of visibility admissible expressions presented above.

Definition 31 (Legal assignment) *If f is declared in class T, then the assignment $o.f = exp$ is* legal *iff*

- $T < o.inv$, *and*
- *for each class T' in the* `dependent`*-clause of f and for each access expression* $\texttt{this}.g_1 \ldots g_n.f$ *of kind (2) in Definition 30 in an invariant declared in T':*

$$\forall t \mid t.g_1 \ldots g_n = o \bullet T' < t.inv$$

4.2.3 Translation

Definition 32 (Translation) *The declaration of a field f is translated as follows:*

$$Tr\,(\;T\,f;\;) \quad \equiv \quad T\,f\;\; \texttt{dependent}\; T_1, \ldots, T_n\,;$$

where $T_1, \ldots, T_n$ are all the classes whose invariants contain access expressions $\texttt{this}.g_1 \ldots g_n.f$ *of the form (2) in Definition 30.*

The declaration of a method m in class T is translated as follows:

$$
Tr\,(\; m(S\,x)\;\{\; Body \;\}\;) \quad \equiv \quad
\begin{array}{l}
m(S\,x) \\
\quad \textbf{requires}\; \texttt{this}.owner.typ < \texttt{this}.owner.obj.inv \wedge \\
\qquad\qquad \forall o \mid arePeers(o, \texttt{this}) \bullet o.inv = type(o) \\
\quad \textbf{ensures}\; \forall o \mid arePeers(o, \texttt{this}) \;\bullet\; o.inv = type(o) \wedge \\
\qquad\qquad\qquad o.owner.typ < o.owner.obj.inv \\
\quad \{ \\
\qquad \textbf{foreach}\; o\; \textbf{with}\; arePeers(o, \texttt{this})\;\; \textbf{do} \\
\qquad\quad \textbf{fullyunpack}\; o \\
\qquad Tr\,(Body) \\
\qquad \textbf{foreach}\; o\; \textbf{with}\; arePeers(o, \texttt{this})\;\; \textbf{do} \\
\qquad\quad \textbf{fullypack}\; o\; \textbf{at}\;\; T \\
\quad \}
\end{array}
$$

The declaration of the constructor in class T is translated as follows:

$$
Tr\,(\; T(S\,x)\;\{\; Body \;\}\;) \quad \equiv \quad
\begin{array}{l}
T(S\,x) \\
\quad \textbf{requires}\; \forall o \;\mid\; o \neq \texttt{this} \wedge arePeers(\texttt{this}, o) \;\bullet\; o.inv = type(o) \\
\quad \textbf{ensures}\; \texttt{this}.owner.typ < \texttt{this}.owner.obj.inv \wedge \\
\qquad\qquad \texttt{this}.inv = T \wedge \\
\qquad\qquad \forall o \;\mid\; o \neq \texttt{this} \wedge arePeers(\texttt{this}, o) \;\bullet\; o.inv = type(o) \\
\quad \{ \\
\qquad Tr\,(Body) \\
\qquad \textbf{pack}\; \texttt{this}\; \textbf{as}\;\; T \\
\quad \}
\end{array}
$$

A method call $o.m(x)$ appearing in class T is translated as follows:

$$
\begin{array}{lcl}
\mathit{Tr}\,(& & \textbf{if } \mathit{arePeers}(o, \texttt{this}) \textbf{ then} \\
\quad o.m(x) & \equiv & \quad \textbf{foreach } o' \textbf{ with } \mathit{arePeers}(o', \texttt{this}) \textbf{ do} \\
) & & \quad\quad \textbf{fullypack } o' \textbf{ at } T \\
& & o.m(x) \\
& & \textbf{if } \mathit{arePeers}(o, \texttt{this}) \textbf{ then} \\
& & \quad \textbf{foreach } o' \textbf{ with } \mathit{arePeers}(o', \texttt{this}) \textbf{ do} \\
& & \quad\quad \textbf{fullyunpack } o'
\end{array}
$$

4.2.4 Translation's correctness

Theorem 6 *Let $\boldsymbol{P}$ be a JML program that is type correct in the Universe Type System. If $\boldsymbol{P}$ has legal assignments, admissible invariants and legitimate JML executions, then $\mathit{Tr}(\boldsymbol{P})$ is a* Spec♯ *program with legal assignments, admissible invariants, and legitimate executions.*

Proof. Every JML legal assignment in $\mathbf{P}$ is, in particular, a Spec♯ legal assignment in $\mathit{Tr}(\mathbf{P})$. Let $o.f = \mathit{exp}$ be an arbitrary legal assignment in $\mathbf{P}$, where field f is declared by a class T. By Definition 29, we get $\mathit{arePeers}(o, \texttt{this})$, where `this` is the receiver object of the method that contains the assignment. Assuming that $\mathit{type}(o)$ is visible in m, the statement **fullyunpack** o is executed before the assignment (see Definition 32). Consequently, $o.\mathit{inv} = \texttt{Object}$. It then obviously holds $T < o.\mathit{inv}$. Let us now assume that there exists a class T' in the `dependent`-clause of f. Let us consider an arbitrary access expression $\texttt{this}.g_1 \ldots g_n.f$ of kind (2) from Definition 30 in an invariant declared in T' such that $t.g_1 \ldots g_n = o$. As the fields $(g_i)_{i=1}^{n}$ are `peer` fields, t and o are peers, *i.e.*, $\mathit{arePeers}(t, o)$. By this, $\mathit{arePeers}(o, \texttt{this})$ and Lemma 3, we get $\mathit{arePeers}(o, \texttt{this})$. Assuming that $\mathit{type}(t)$ is visible in m, we get that $t.\mathit{inv} = \texttt{Object}$ upon executing the statement **fullyunpack** t at the beginning of m (see the translation of a method declaration). Therefore, $T' < t.\mathit{inv}$. Hence, according to Definition 31, the assignment $o.f = \mathit{exp}$ is legal in $\mathit{Tr}(\mathbf{P})$.

Every JML admissible invariant (Definition 28) is, in particular, a Spec♯ admissible invariant (Definition 25). That is because every visibility admissible access expression in JML is, in particular, a visibility admissible expression in Spec♯. To show that, let us consider an arbitrary visibility admissible access expression $\texttt{this}.g_0 \ldots g_n$ appearing in the invariant of a JML class T. By Definition 27, the field g_0 is declared by T. We distinguish the following three cases:

- $n = 0$. The access expression $\texttt{this}.g_0$ is an expression of the form (1) from Definition 30. Thus, it is visibility admissible in Spec♯.
- every field $(g_i)_{i=0}^{n}$ is a `peer` field. By Definition 32, in the resulting Spec♯ program, class T will appear in the `dependent`-clause of field g_n. Consequently, the access expression is visibility admissible in Spec♯ according to Definition 30 (expression of form (2)).
- $n > 0$ and g_0 is declared as a `rep` field. In this case, the access expression is in Spec♯ of the form (1) in Definition 30. So, the expression is visibility admissible in Spec♯.

$\mathit{Tr}(\mathbf{P})$ executions' legitimateness can be proved by induction on the execution's length. As this proof is similar to the corresponding proof part from Theorem 4, we omit it here. □

References

[1] The Java Modeling Language. Available at `http://www.eecs.ucf.edu/~leavens/JML/`.

[2] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of Object-oriented Programs with Invariants. *Journal of Object Technology*, 3(6):27–56, 6 2004. Special issue: ECOOP '03 workshop on Formal Techniques for Java–like Programs.

[3] Microsoft Corporation. The Spec# Programming System. Available at `http://research.microsoft.com/specsharp/`, 2004.

[4] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer-Verlag, 2004.

[5] P. Müller and A. Rudich. Ownership transfer in Universe Types. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2007.

[6] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular Invariants for Layered Object Structures. *Science of Computer Programming*, 62(3):253–286, 2006.