Formal encoding of JML Level 0 specifications in Jive

Ádám Darvas and Peter Müller

ETH Zurich, Switzerland
{adam.darvas,peter.mueller}@inf.ethz.ch

$Date: 2007/04/30 11:58:26 $Revision: 1.12

Abstract

This report describes a formal encoding of the most important JML specification constructs to first-order logic. While the translation gives a general way of handling these constructs, the report is based on the underlying programming logic and theorem prover of the Jive system.

1 Introduction

Our group, together with the Softwaretechnik group at TU Kaiserslautern, work on the development of Jive, the Java Interactive Verification Environment [24]. The tool enables users to interactively prove properties of Diet Java Card (DJC) programs annotated with Java Modeling Language (JML) specifications [17]. DJC is a subset of Java Card, providing all important object-oriented features like inheritance and dynamic binding of methods. JML is a behavioral interface specification language specifically tailored to Java. Its main purpose is to be easily comprehensible by Java programmers and at the same time be capable of rigorously specifying various aspects of programs.

The development of specification languages showed that while the bridging of declarative specification languages (e.g. VDM, Z) to verification tools is relatively easy and well-understood, they require considerable learning effort from non-expert users.

Another approach is to keep the specification language as close to the programming language as possible, thus enabling programmers to easily specify their own programs. While this makes such languages (e.g. Eiffel, Spec#, JML) practical, the bridging to a verification tool is more subtle since the gap between a programming and a specification language is very wide.

Though many aspects of such specification languages are well-understood for the research community, there is still a considerable amount of research work going on to clarify certain issues and to assign them proper formal semantics.

In this report we give translations for the most basic constructs of JML: the constructs that are classified as JML Level 0 elements by the community [18, Sec. 2.9.1]. The translation we describe here defines the way proof obligations, in the form of Hoare-triples, are generated from JML specifications in Jive.

The report is organized as follows. Section 2 gives a short introduction to JML Level 0, the subset of JML that we handle in this report. Section 3 describes the elements of the underlying programming logic of Jive that are relevant for the understanding of this report. Section 4 gives the translation of all JML Level 0 elements that Jive supports. Finally, we list related work and issues that we plan to address in the future.

2 JML Level 0

In this section we give a brief overview of all elements of JML Level 0 and refer the reader to [18] for a full account—Section 2.9.1 contains the exact description of the different language levels.

JML modifiers

JML Level 0 extends Java's set of modifiers with **spec_public**, **spec_protected**, **nullable**, **instance**, **model**, **ghost**, and **helper**.

The first two modifiers allow members to appear in specifications in which otherwise they would be forbidden to appear due to their declared Java visibility. For instance, a private field may appear in a public postcondition if and only if the field has the additional JML modifier **spec_public**.

The semantics of JML does not, by default, allow declared fields, method parameters, and return values of reference type to be `null`. To allow them to take the `null` value, one has to use the **nullable** modifier in field declarations and method signatures.

The **instance** modifier is used to declare that a ghost or model member is not static but bound to instances. For example, a model field of an interface can be declared to be an **instance** field. Without the modifier the field would be a static member according to Java.

The **model** modifier can be used at field declarations. Model fields are fields that are only used for specification purposes. They are treated as ordinary fields, but they cannot occur in implementations. Their values are not looked up from the heap, but are determined by the values of other (concrete or model) fields, as specified by the represents clause attached to a given model field.

The **ghost** modifier can be used at field declarations. Ghost fields are similar to model fields in that they can only be used for specification purposes. However, their values can be implicitly set in implementations by the use of JML's **set statements**.

In the rest of this report we will refer to ordinary fields as *concrete fields* and to model and ghost fields as *abstract fields*.

Private methods and constructors can be declared with the **helper** modifier. Such methods and constructors are allowed to violate declared invariants.

Type specifications

Type specifications are specifications that are declared on the type level. In JML Level 0 these are object **invariants**, **represents** clauses for model fields, **initially** clauses, and **\TYPE** specifications.

Object invariants specify the consistent states of objects. In JML, they specify properties that must hold in all *visible states*, that is, in all pre- and post-states of method executions [28]. This invariant semantics is called the *visible state semantics*. Level 0 only supports *instance* invariants (i.e. no static invariants). Invariants of supertypes are inherited by subtypes.

The relation between the value of a model field and the values of other (possibly abstract) fields is determined by the **represents** clause. JML Level 0 only allows the *functional* form of the clause, meaning that the mapping has to be a function, in contrast to a relation.

An **initially** clause allows one to specify properties about the initial values of fields. That means, non-helper constructors must establish the predicate given in the clause.

JML provides the special type specification **\TYPE** which represents the kind of all Java types [18]. It can be used, for instance, to declare fields whose value is a Java type.

Method specifications

Method behavior is specified by the use of preconditions, postconditions, exceptional postconditions, and assignable clauses. The precondition (**requires** clause) of a method specifies what must hold when calling the method. The postcondition (**ensures** clause) specifies what must hold if and when it terminates. The exceptional postcondition consists of two clauses: (1) the **signals_only** clause specifies what type of exceptions may be thrown by the method; (2) the **signals** clause specifies what must hold at the point when a certain type of exception is thrown. The **assignable** clause of a method specifies which locations the method may modify. All other variables have to remain unchanged.
Overriding methods inherit all visible method behavior specifications given in supertypes.
Specification cases. JML provides different kinds of **specification cases** of which the most general one is the *heavyweight behavior specifications case*. It may contain all five clauses mentioned above. The other specification cases are either just syntactic sugars or define different default values for omitted specification clauses.

Data groups. In order to specify assignable clauses in a modular way, **data groups** [21] can be used in JML. They allow one to gather concrete and abstract fields into a set. When a model field is declared, a data group with the same name is created and the declared model field is always member of that data group. In JML Level 0, additional members can be added by so-called static inclusion using the **in** clause [21]. For instance, assume `color` is a data group, then the field declaration "**int** `RGB` **in** `color;`" means that field `RGB` is a member of the data group.

Data groups are used in assignable clauses: a data group appearing in an assignable clause specifies that the method in hand may modify the elements of the data group. In the example above, if `color` appears in the assignable clause of a method, it means that the method may modify the `RGB` field of the **this**-object.

JML expressions

JML's expression syntax extends Java's side-effect free expression syntax. The most important extensions are additional logical connectives (e.g. implication, equivalence), quantifiers, the "**\old**" construct used to refer to values in pre-states, and the "**\result**" construct which refers to the return value of non-void methods. An important restriction of JML Level 0 is that methods and constructors are not allowed to be called from within specification expressions.

The list of JML Level 0 expressions that are handled by our translation and JIVE is given in Section 4.4.

JML statements and annotation statements

For verification systems the handling of loops require a *loop invariant*. Loop invariants are typically given by the programmer or by the user of the verification system. The former case is supported by JML: using the **maintaining** keyword a loop invariant can be attached to loops.

The **assert** statement requires the given predicate to hold, while the **assume** statement specifies that the given predicate holds. Ghost fields can be assigned to via **set statements**, in the form "**set** g = 5;" where g is a ghost field.

Specification library

JML supports specifications to be placed in `.spec` files. This is useful when the specified compilation unit is only available in binary form or the source file is not to be modified. For instance, the specification of the Java API library is given in `.spec` files. This allows one to verify programs that contain calls to the API or other library code.

Universe modifiers

JML allows one to specify ownership of objects [8]. JML implements the Universe type system [27, 11]. JML Level 0 supports the two modifiers **rep** and **peer**. A field f declared with the **rep** modifier in object o means that f is either `null` or the object referred to by f is owned by o. A **peer** modifier would mean that f is either `null` or o and the object referred to by f have the same owner object.

3 Logical Background of JIVE

To formalize properties of the object store, we use the store model of Poetzsch-Heffter and Müller's program logic [30]. It is formalized in multi-sorted first order logic with recursive datatypes.

Types and Values. Java's types and values are modeled by the sorts *Type* and *Value*, respectively. Sort *Type* contains primitive types, the type of the `null` reference, and class types. The reflexive, transitive subtype relation is denoted by $\preceq$. A *Value* is a value of a primitive type, the `null` reference, or a reference to an object. The function $typeof : Value \to Type$ yields the type of a value.

Object States. Object states are modeled via *locations* (instance variables). For each field of its class, an object has a location. Depending on whether the field is concrete or abstract, the location is a concrete or an abstract location, respectively. The sort *FieldId* is the sort of unique field identifiers of a program. The function $loc(X, f)$ yields the location of the object referenced by X for field f, or *undefined* if the object does not have a location for f. Conversely, $obj(L)$ yields a reference to the object a location L belongs to. For brevity, we write $X.f$ for $loc(X, f)$ in the following.

$$loc : Value \times FieldId \rightarrow Location \qquad obj : Location \rightarrow Value$$

Since the properties of these functions are not needed in this report, we refer the reader to [30] for their axiomatization.

Object Stores. Object stores are modeled by an abstract data type with main sort *Store* and operations to read and update locations, to create new objects, and to test whether an object is allocated. Poetzsch-Heffter and Müller present these functions and their axiomatization in [30].

In this paper, we need the following store operations: $OS\langle T\rangle$ yields the object store that is obtained from OS by allocating a new object of class T. $new(OS, T)$ yields a reference to this object. $OS(L)$ denotes the value held by location L in store OS. $alive(X, OS)$ yields true if and only if object X is allocated in OS. Values of primitive types are allocated in all stores. The sort *ClassId* is the sort of unique class identifiers of a program.

$$\_\langle\_\rangle : Store \times ClassId \rightarrow Store \qquad new : Store \times ClassId \rightarrow Value$$
$$\_(\_) : Store \times Location \rightarrow Value \qquad alive : Value \times Store \rightarrow Bool$$

The constant symbol \$ of sort *Store* is used to refer to the current object store in formulas. It can be considered as a global variable.

Logic of Jive. Jive implements a Hoare-style logic [31] which is based on the store model sketched above. The details of the logic is not relevant for this report. Here we only introduce the special variables of the logic that our translation uses too [12].

The logical variable χ tracks the *status* of the method being verified. The variable can take two values: *normal* in case of normal termination of the method or *exc* in case an exception has been thrown. The program variable *excV* holds the reference to the exception object that was thrown. Otherwise, its value may be undefined.

The program variable *resV* holds the return value of the method in hand. Its value may be undefined in a state where the method did not return yet.

Work-flow of Jive. To give a better understanding of how the store model and the logic of Jive work together we sketch the way one works with Jive.

First the program and its specification is typechecked by the JML compiler. Then proof obligations, in the form of Hoare-triples, are generated and passed to the program prover component. Furthermore, program dependent theories are generated that provide information, for instance, on type relations and attribute declarations to the back-end theorem provers.

The program prover component implements a partial-correctness Hoare-style logic which formalizes the axiomatic semantics of DJC. Typically, a proof obligation is proved in two steps. First, all DJC program constructs are eliminated interactively, but possibly fully automatically with the help of strategies. After this step, the result is a program independent first-order formula. Next, this formula is passed to one of the back-end theorem provers: the interactive Isabelle [29], or the fully automated Simplify [10]. The proof attempt for both provers is supported by the program dependent and program independent theories. While the former is generated on-the-fly, the latter comes with the Jive distribution and formalizes the programming logic of Jive.

4 Translation of JML Level 0 elements

This section describes the translation of the elements of JML Level 0 introduced in Section 2. It also lists the elements that are not supported by this translation, and thus by Jive.

Throughout this report we use the γ function which translates valid JML expressions into terms of the underlying logic of JIVE. The function is described in more details in Section 4.4. For the moment, the reader can assume that the function translates expressions to first-order terms conforming to the programming logic described in Section 3.

4.1 Translating JML modifiers

There is no translation performed on modifiers **spec_public** and **spec_protected**. JIVE builds on top of the JML compiler which performs the visibility checks of specifications including these two modifiers.

In JIVE, if a field f is declared in type T without the **nullable** modifier then the predicate $this.f \neq null$ is conjoined to the invariant of the class (see Section 4.2.1). A method parameter p without the **nullable** modifier results in the predicate $p \neq null$ conjoined to the precondition of the method. A return value without the modifier yields the predicate $resV \neq null$ conjoined to the postcondition of the method (see Section 4.3.1).

JIVE does not support static members, which (among others) prohibits the declaration of fields in interfaces as they are implicitly static members according to Java. However, JIVE supports the **instance** modifier which means that ghost and model members may be declared in interfaces.

Model fields are important means for providing abstract specifications. Clients do not have to (and should not) know about the internal implementation details when writing or reading JML specifications of a type. The two main issues about model fields are: how to encode their represents clauses and how to handle them in assignable clauses. These points are described in Section 4.2.2 and Section 4.3, respectively.

Ghost fields are very similar to concrete fields as their values are read from the store and explicitly set by JML's **set statement**. Thus, our translation handles ghost fields as concrete ones and turns each **set statement** into plain assignment statements.

The handling of the **helper** modifier is discussed in Section 4.2.1.

4.2 Translating type specifications

4.2.1 Translating invariants

In general, invariants are added to pre- and postconditions of methods. There are two exceptions: (a) for constructors the invariant of the object in hand is only added to postconditions; (b) for **helper** methods and constructors invariants are added neither to preconditions nor to postconditions.

Formalization of invariants

In JIVE, invariants are formalized using three kinds of functions.

For each type, T_i, of the program there is a function, Inv_{T_i}, declared with signature $Value \rightarrow Store \rightarrow Bool$. Function $Inv_{T_i}(X, OS)$ yields true if and only if the declared invariant of T_i holds for the T_i-object X in store OS. It is defined as follows:

$$Inv_{T_i}(X, OS) = typeof(X) \preceq T_i \ \Rightarrow \gamma(I_{T_i})[OS/\$, X/\mathbf{this}]$$

where I_{T_i} denotes the specified invariant of type T_i. Note that if X is not subtype of T_i, the function yields trivially true.

Another function, INV, is declared to conjoin the Inv_{T_i} functions. Its signature is $Store \rightarrow Bool$ and $INV(OS)$ yields true if and only if every object that is alive in store OS fulfills its declared and inherited invariants.

$$INV(OS) = \forall\ X{:}Value.\ alive(X, OS)\ \wedge\ X \neq null \Rightarrow \bigwedge Inv_{T_i}(X, OS)$$

Note that this formula enforces inheritance of invariants since the right-hand side of the implication conjoins the Inv_{T_i} functions of all types of the program at hand. Thus, every object X of its actual type

T has to satisfy the invariants declared in T and inherited from supertypes of T. As mentioned above, for all other types T_i that are not supertypes of T, the Inv_{T_i} functions yield true.

For the handling of constructors a third function, *INVC*, is declared with signature $Store \rightarrow Value \rightarrow Bool$. It is very similar to *INV* but exempts the object given as parameter from satisfying its invariants:

$$INVC(OS, Y) = \forall\ X{:}Value.\ X \neq Y\ \wedge\ alive(X, OS)\ \wedge\ X \neq null \Rightarrow \bigwedge Inv_{T_i}(X, OS)$$

Generated proof obligations

These declared functions are used in proof obligations. When generating Hoare-triples for non-helper methods, formula *INV*(\$) is conjoined to pre- and postconditions. For methods and constructors declared with **helper** modifier, none of the functions are conjoined to pre- or postconditions.

init methods. As DJC, the input language of JIVE, only allows default constructors, a workaround is needed to mimic non-default constructors: right after a new T-object is created using the default constructor, the special *init*T method need be called on the new object to initialize it.

These *init* methods are treated in a special way by the proof obligation generator: $INVC(\$, this)$ is conjoined to their precondition which prevents assuming that the invariant of the newly created object holds before being initialized. To the postcondition *INV*(\$) is conjoined requiring the initialized object to satisfy its invariant. Furthermore, the following conjunct is added to the precondition of *init* methods:

$$(\forall\ loc{:}Location.\ \$(L) \neq this)\ \wedge\ (\forall\ f{:}Field.\ \$(this.f) = init(typeof(f)))\ \wedge\ param_i \neq this$$

which expresses that (1) no location references the **this**-object, (2) fields of the **this**-object hold their initial values and that (3) the parameters of the *init* method, if any, do not reference the **this**-object. This conjunct expresses properties one can assume when entering a constructor and is needed in order to verify *INV*(\$) in the postcondition of the method.

However, it must be noted that it is the users responsibility to use *init* methods appropriately, for instance, not to pass the newly created object as parameter, which would invalidate assumption (3). This could possibly lead to unsoundness. We are planning to extend DJC with non-default constructors, which would eliminate this problem in JIVE.

4.2.2 Translating represents clauses

Model fields can only occur in specifications and their values are determined by represents clauses. These clauses give the relation between the value of a model field and the values of ordinary fields and other model fields.

In JIVE, represents clauses are turned into axioms. Assume the following JML code in class T:
model int m;
represents m <− expr;
where expr is an arbitrary expression of type int. The corresponding axiom that is generated in JIVE is the following:

$$OBJ \neq null\ \wedge\ alive(OBJ, OS)\ \wedge\ typeof(OBJ)\ \preceq T \Rightarrow OS(OBJ.m) = \gamma(expr)[OBJ/\mathbf{this}, OS/\$]$$

where *OBJ* is of sort *Value*, and *OS* is of sort *Store*. The axiom establishes the relationship expressed by the represents clause for allocated T-objects.

Currently JIVE does not ensure soundness of the generated axioms. There are mainly two ways the axioms can introduce unsoundness

1. A program can contain multiple represents clauses for the same model field. These clauses can contradict each other, for instance, one clause specifying the value of **this**.m to be 5 and an other clause (e.g. in a subtype) specifying it to be 10. The corresponding two axioms that would be generated lead to unsoundness as $5 = 10$, and thus *false* can be derived from them.

behavior
  requires P;
  ensures Q;
  signals_only $\mathrm{ET}_1, ..., \mathrm{ET}_n$;
  signals (ET_j e) R;
  assignable $\mathrm{loc}_1, ..., \mathrm{loc}_n$;
T m(S p) { ... }

Figure 1: General form of a heavyweight behavior specification case.

To avoid multiple axioms to contradict each other, one could restrict specifications to contain only one represents clause per model field. Since JML Level 0 only allows the functional form of the clause, this restriction seem to cause problems only in the case of data refinement in subtypes.

To avoid unsoundness stemming from an unsatisfiable represents clause (e.g. m <− m + 1), one can require the proof of the existence of a possible value (a *witness*) that satisfies the represent clause specified for a given model field. This kind of solution has been applied to model fields [22] and method calls in specifications [9].

2. The represents clause of a model field is typically well-defined only in case the invariant of the object at hand holds [22]. A technique based on the Boogie methodology [2] that ensures that the values of model fields are consistent with their specified represents clauses is described in [22]. However, the Boogie methodology is not based on the visible state semantics. We are not aware of a technique that solves this problem in the visible state semantics.

4.2.3 Translating initially clauses

In general, the predicate of the initially clause must be established by all constructors that are subtypes of the enclosing type. Since Jive uses the special *init* methods instead of non-default constructors, the predicate of the clause need be added to the postconditions of the *init* methods of the enclosing type and subtypes. Initially clauses are currently not supported, mainly because it is not a frequently used feature of JML.

4.2.4 \TYPE type specification

The **\TYPE** type specification is not supported by Jive. This is due to the underlying logic of Jive in which Values and Types are two distinct sorts. Thus, for instance, a field declared with **\TYPE** would need to yield a Type when accessed. However, the logic of Jive encodes field accesses as yielding Values.

4.3 Translating method specifications

4.3.1 Translation of method specification clauses

The most general method behavior specification construct of JML is the heavyweight behavior specification case. This construct may contain preconditions, postconditions, exceptional postconditions and assignable clauses.

It is sufficient to give the translation of this kind of specification case as the other specification cases are either just syntactic sugars (the heavyweight normal and exceptional behavior specification cases) or mean just different default values for omitted clauses (the lightweight specification case) [32]. At the end of this subsection we discuss how these other specification cases are handled.
The general form of the heavyweight behavior specification case is shown on Figure 1, where

- P, Q, and R are boolean JML expressions,
- ET_i are subtypes of type Exception,

- $\mathrm{ET_j}$ is a subtype of type Exception and e is an object of type $\mathrm{ET_j}$. If R does not contain e then it can be omitted from the clause,
- $\mathrm{loc_i}$ are concrete or abstract locations.

The translation of behavior specification cases yields Hoare-triples as described below.

Precondition. The precondition P is translated by the γ function and the resulting logical formula is conjoined to the precondition of the Hoare-triple.

Postcondition. The postcondition Q is translated by the γ function and the resulting logical formula is conjoined to the postcondition of the Hoare-triple. However, since the specified postcondition has to hold only in case the method terminated normally, the conjoined formula is $\chi = \mathit{normal} \Rightarrow \gamma(\mathrm{Q})$.

Exceptional postcondition. The exceptional postcondition R is translated by the γ function and the resulting logical formula is conjoined to the postcondition of the Hoare-triple. However, the exceptional postcondition has to hold only in case the method terminated abruptly and the type of the exception is subtype of the specified type. Thus, the conjoined formula is

$$\chi = \mathit{exc} \Rightarrow (\mathit{typeof}(\mathit{excV}) \preceq \mathit{ET}_j \Rightarrow \gamma(\mathrm{R})[\mathit{excV}/\mathrm{e}]).$$

Signals-only clause. The **signals_only** clause specifies what type of exceptions the method is allowed to throw. This is expressed by the following formula conjoined to the postcondition:

$$\chi = \mathit{exc} \Rightarrow (\mathit{typeof}(\mathit{excV}) \preceq \mathit{ET}_1 \vee \ \ldots \ \vee \ \mathit{typeof}(\mathit{excV}) \preceq \mathit{ET}_n)$$

Assignable clause. The assignable clause of a method defines a set of locations. The locations can be both concrete and abstract. As already mentioned in Section 2, model fields can appear in assignable clauses with the semantics that any concrete or abstract field that is element of the data group corresponding to the model field may be modified.

The standard technique to encode this semantics is to calculate the *downward closure* on the set of concrete and abstract locations listed in the assignable clause [21]. The downward closure of the assignable clause yields the set of all locations that may be modified by the method. It is calculated by recursively adding all fields to the set that are elements of a data group that is already an element of the set. The downward closure applied to a set of locations, L, is denoted by $\delta(L)$.

The semantics of JML does not allow locations that are not mentioned in the assignable clause to be modified even temporarily.

Assignable clause in Jive. In Jive, the handling of assignable clauses differ in two ways from that of JML's semantics.

- our translation allows temporary modifications of locations that are not listed in the clause. This means that our translation only requires values of locations not listed in the clause to be the same in pre- and post-states. This is due to the lack of some kind of *throughout* modality in the logic.
- our translation applies a more liberal semantics: fields of objects that are not peers to the receiver object may be modified without being mentioned in the assignable clause. This liberal semantics is explained as follows. (1) The representation of the receiver object and any other objects should not be of concern to other objects, thus they should not be concerned about changes in the representation either. (2) The modification of a field, for instance, in the receiver object might cause the modification of a model field in an object that is "higher up" in the ownership hierarchy. This can happen if the model field depends on the field of the object, that is, the field is mentioned in the represents clause of the model field. However, tracking such dependencies is very difficult and requires complicated techniques to handle in a sound way [20, 26]. Our approach is to use a simple over-approximation. If one needs stronger guarantees then stronger postconditions need be written.

$\{INV(\$) \ \wedge \ \gamma(\mathrm{P}) \ \wedge \ \$ = S \ \wedge \ M = \delta(\{\mathrm{loc_1}, ..., \mathrm{loc_n}\}) \ \wedge \ p \neq null\}$
m(S p)
$\{INV(\$) \ \wedge \ (\chi = normal \Rightarrow \gamma(\mathrm{Q})) \wedge$
$\chi = exc \Rightarrow (typeof(excV) \preceq ET_1 \vee \ \ldots \ \vee \ typeof(excV) \preceq ET_n) \wedge$
$(\chi = exc \Rightarrow (typeof(excV) \preceq ET_j \Rightarrow \gamma(\mathrm{R})[excV/\mathrm{e}])) \wedge$
$\forall$ *loc*:*Location*. $(loc \in M \ \vee \ \neg alive(obj(loc), S) \vee$
$\$(this.owner) \neq \$(obj(loc).owner) \ \vee \ \$(loc) = S(loc))$

Figure 2: Hoare-triple corresponding to the general form of behavior specification case shown on Figure 1.

The translation of assignable clauses has two components. First, the formula

$$S = \$ \ \wedge \ M = \delta(\{\mathrm{loc_1}, ..., \mathrm{loc_n}\})$$

is conjoined to the precondition of the triple, where the first conjunct saves the pre-store in the logical variable S, and the latter saves the set of modifiable locations in the logical variable M.
Second, the formula

$$\forall \ loc{:}Location. \ (loc \in M \ \vee \ \neg alive(obj(loc), S) \ \vee \ \$(this.owner) \neq \$(obj(loc).owner) \ \vee \ \$(loc) = S(loc))$$

is conjoined to the postcondition of the triple, which expresses that a locations is either (1) member of the set of modifiable locations, or (2) the object corresponding to the location did not exist in the pre-state, or (3) the object corresponding to the location is not peer of the receiver object, or (4) the value of the locations is unchanged.

JIVE currently only uses ownership type information (using the **rep** annotation) for this slightly modified assignable clause semantics. The consistency of the ownership structure must be enforced either by a type system or by proof obligations to make the above semantics useful. Currently JIVE does not enforce the consistency either way. We plan to use the Universe type system in the near future which is integrated into the JML compiler [11].

Putting it all together. Putting these items all together, the general form of behavior specification cases shown on Figure 1 translates to the Hoare-triple shown on Figure 2.[1]

4.3.2 Translation of multiple clauses, defaults, and sugars

In this section we show how multiple specification clauses, defaults, and sugars are treated by our translation. The explanations (e.g. "second conjunct of the precondition") refer to the triple on Figure 2.

Multiple specification clauses

In case of multiple **requires** clauses

requires $\mathrm{P_1}$;
...
requires $\mathrm{P_n}$;

the specification expressions $\mathrm{P_i}$ are simply conjoined in the second conjunct of the precondition: $\gamma(\mathrm{P_1}) \wedge \ldots \wedge \gamma(\mathrm{P_n})$. Multiple **ensures** clauses are handled analogously. In case of multiple **signals** clauses

signals ($\mathrm{E_1}$ $\mathrm{e_1}$) $\mathrm{R_1}$;
...
signals ($\mathrm{E_k}$ $\mathrm{e_k}$) $\mathrm{R_k}$;

the fourth conjunct of the postcondition is of the form

[1] Reminder: conjunct $p \neq null$ is added to the precondition because method parameters are non-null by default.

$$\chi = exc \Rightarrow ((typeof(excV) \preceq \mathrm{E}_1 \Rightarrow \gamma(\mathrm{R}_1)[excV/\mathrm{e}_1]) \wedge \ldots \wedge (typeof(excV) \preceq \mathrm{E}_\mathrm{k} \Rightarrow \gamma(\mathrm{R}_\mathrm{k})[excV/\mathrm{e}_\mathrm{k}]))$$

This means

- nothing has to be proven on the post-state if the type of the exception that was thrown is not in subtype relation with any of the types E_i,
- if the type of the exception that was thrown is in subtype relation with multiple types E_i then multiple R_i postconditions have to be proven.

Multiple behavior specification cases of a method lead to multiple proof obligations.

The generated triples of a method m in type C only contains the specifications declared for m in type C. Inheritance of non-private method specifications (i.e. behavioral subtyping) is enforced by the logic of JIVE [31].

Defaults for behavior and lightweight specifications.

JML does not specify the defaults for omitted specification clauses of lightweight specification cases. On the other hand, heavyweight specification cases have fixed defaults. Our translation assigns these defaults to omitted clauses of lightweight specification cases.
Here we give these defaults and describe how they modify the generated proof obligations:

- an omitted **requires** clause means an implicit "**requires true**;" clause. That is, the second conjunct can be omitted from the precondition.
- an omitted **ensures** clause means an implicit "**ensures true**;" clause. That is, the second conjunct can be omitted from the postcondition.
- an omitted **signals_only** clause defaults to the declared **throws** clause of the method. If the **throws** clause is empty then the method is not allowed to throw any type of exception, i.e. it is equivalent to a normal behavior specification case (see below).
- an omitted **signals** clause means an implicit "**signals** (Exception) **true**;" clause, i.e. the method is considered to be correct in case it throws any exception. This means that the fourth conjunct can be omitted from the postcondition.
- an omitted **assignable** clause means an implicit "**assignable \everything**;" clause. This means that the set of modifiable locations M does not have to be created in the precondition (fourth conjunct) and the fifth conjunct can be omitted from the postcondition.

Desugaring normal and exceptional behavior.

- a normal behavior specification case is syntactic sugar for a behavior specification case to which the "**signals** (Exception) **false**;" clause is added. This can be expressed by conjoining "$\chi \neq exc$" as the fourth conjunct in postconditions.
- an exceptional behavior specification case is syntactic sugar for a behavior specification case to which clause "**ensures false**;" is added. This can be expressed by conjoining "$\chi \neq normal$" as the second conjunct in postconditions.

Desugaring purity.

To specify that a method may not modify any pre-exisiting locations, the "**assignable \nothing**" clause can be used.[2] In this case there is no need to use logical variable M. Thus, the fourth conjunct can be omitted from the precondition and the fifth conjunct of the postcondition simplifies to

$$\forall\ loc{:}Location.\ alive(obj(loc), S) \Rightarrow \$(loc) = S(loc).$$

Note that this translation reflects *weak purity*, that is, locations of objects that do not exist in the pre-state might be modified freely.

[2]This can be specified by the **pure** annotation too, but it is not part of JML Level 0.

4.4 Translating JML expressions

The expressions that can be used in JML predicates are an extension of the side-effect free Java expressions. Note that in Level 0 calls to pure methods and constructors are disallowed in specification expressions.

In this section we first look at the translation of Java expressions. In particular, we discuss how JIVE handles abrupt termination of expressions. Then we give the translation of expressions that JML Level 0 adds to the Java expression syntax.

4.4.1 Java expressions

Most of the expressions that may appear in JML specifications can be translated to first-order logic formulas in a straight-forward way. However, there are some cases where the translation is not obvious.

Abrupt termination and underspecification

Since JML extends the side-effect free expression syntax of Java, predicates may throw exceptions. In this case, according to the semantics of JML, an arbitrary value of the normal return type may be picked. Our aim is to model this in JIVE.

Our approach is to leave such expressions *underspecified* following the approach described in [15]. That is, in case an expression would throw an exception we leave its value underspecified—the expression has a value, but we do not know which. Note that this approach allows us to use a two-valued logic and theorem-prover.

In preconditions this means that the predicate which is left underspecified cannot be used as knowledge assumed. However, this does not mean that such a predicate would doom the whole precondition underspecified or the whole proof obligation unprovable. As an example, consider the following triple:

$$\{this.f = 0 \ \wedge \ null.f = 0\} \ \ \textbf{skip} \ \{this.f = 0\}$$

The expression $null.f$ throws an exception at runtime. The programming logic of JIVE presented in Section 3 yields an undefined value for the expression. However, after applying the Hoare-rule for statement **skip** (the empty statement), the resulting formula to prove is

$$this.f = 0 \ \wedge \ null.f = 0 \ \Rightarrow \ this.f = 0$$

which is provable independently of the truth value of $null.f$.

In postconditions this means that the predicate which is left underspecified cannot be proved. Again, this does not mean that the postcondition is underspecified or the triple is unprovable. Similarly to the previous example, the triple

$$\{this.f = 0\} \ \ \textbf{skip} \ \{this.f = 0 \ \vee \ null.f = 0\}$$

remains provable.

In JIVE, a specification expression might terminate abruptly due to: (1) a field access $o.f$ when o is `null`, (2) a division or modulo by zero, (3) a cast $(T)E$ when the type of expression E is not (sub)type of T.

Note that the input language of JIVE does not support arrays, thus no abrupt termination due to array accesses need be handled.
Here we describe how we render the above three cases to an underspecified value.

Field access. Due to the underlying programming logic of JIVE, an attempt to access field $o.f$ when o is `null` leads to an underspecified value.

Division and modulo by zero. Currently JIVE translates the division and modulo operators directly to Isabelle's corresponding `div` and `mod` operators. This is not in line with the semantics of Java. For instance, Isabelle's `div` operator yields zero in case the divisor is zero while in Java an exception would be thrown.

A faithful formalization of Java's arithmetic semantics is given in [33, 34]. A future version of JIVE might build upon that work in order to handle arithmetic formulas according to the semantics of Java (including e.g. arithmetic overflows).

Casts. In the Isabelle prelude we generate an Isabelle function declaration with signature:

$cast : Value \times Type \rightarrow Value$

where the two parameters are the expression being cast and the type of the cast expression. The definition of the function leaves the cast underspecified in case it would throw an exception, otherwise the cast results in the original cast expression. This can be given (as a conservative extension) in Isabelle:

$typeof(X) \preceq T \Rightarrow cast(X, T) = X$

Then, a cast expression $(e_1)\ e_2$ is turned into the Isabelle function application $cast(\gamma(e_2), \gamma(e_1))$.

Basic operators

We list the Java operators whose translation is not straight-forward.
Logical operators. The & and | operators can be translated to logical $\wedge$ and $\vee$ when used in specifications. This is due to the semantics of JML in case of underspecified values. For instance, "**false** & o.f" yields false, even if the field access causes abrupt termination. The same holds for "o.f & **false**" as in JML it is equivalent to the previous predicate.

In specifications one often makes use of the lazy evaluation of the conditional-and (&&) and conditional-or (||) operators, thereby preventing abrupt termination of specification expressions [19]. For instance, consider the precondition "o!=**null** && o.f==10". These operators can also be turned into the logical $\wedge$ and $\vee$ operators. The reason is again the semantics of JML: (1) an underspecified value does not doom the whole predicate to an underspecified truth value; (2) predicates "o!=**null** && o.f==10" and "o.f==10 && o!=**null**" are equivalent.
Conditional operator. The general form of the conditional-operator is "e_b ? e_1 : e_2", where e_b is a boolean expression and in case it evaluates to true then e_1 gives the resulting value, otherwise e_2. This can be directly mapped to Isabelle's predefined if-then-else construct: *if* $\gamma(e_b)$ *then* $\gamma(e_1)$ *else* $\gamma(e_2)$.
For the Simplify integration, a new term has been introduced which is defined in the straight-forward way.
Instanceof operator. The instanceof operator of the general form, *ref* **instanceof** T, can be expressed in our programming logic as $typeof(ref) \preceq ct(T)$, in case T is a concrete type (function ct is explained in Section 4.4.2).

4.4.2 JML extensions to the Java expression syntax

Here we give a list of JML specification expression elements that our translation handles.

- JML allows one to write informal specifications. The meaning of such specifications is not determined by JML. Following [7], JIVE considers them to always hold.
- JML introduces `==>` and `<==` for logical implication, and `<==>` and `<=!=>` for equivalence and inequivalence. They are translated in the straight-forward way.
- JML allows one to write universally and existentially quantified expressions. The syntax of the former is as follows:

  (**forall** T_1 t_1,..., T_n t_n ; e_1 ; e_2)

  The quantification ranges over all potential values of the variables declared which satisfy the *range* predicate e_1. If and only if all these values satisfy predicate e_2, the whole expression yields true. This can be translated by the following pattern:

  $\forall\ t_1{:}T_1, \ldots, t_n{:}T_n.\ (\gamma(e_1) \Rightarrow \gamma(e_2))$.

  The construct and the handling of existential quantification (with keyword **\exists**) is analogous.

- **\fresh(x)** asserts in post-states that x is non-null and that the object bound to identifier x was not allocated in the pre-state. The argument(s) can be any reference type. This is expressed by the formula:

  $$\$(x) \neq \mathit{null} \wedge \neg \mathit{alive}(\mathit{obj}(x), S) \wedge \mathit{alive}(\mathit{obj}(x), \$),$$

  where S is a logical variable for the store in the pre-state.

- an **\old** construct can occur in expressions of **ensures** and **signals** clauses. Our translation takes the whole expression inside the construct and creates a fresh logical variable for it which then can be used in the postcondition.
  For example, the postcondition "a==**\old**(x.f)" would be translated by conjoining term $V = \$(x.f)$ to the precondition and term $a = V$ to the postcondition, where V is a fresh logical variable.
  In JML, if formal parameters occur in **ensures** or **signals** clauses without the usage of **\old**, they still implicitly refer to pre-values, thus such implicit occurrences of pre-values have to be discovered and logical variables have to be created.

- **\result** refers to the resulting value of a non-void method. It is simply translated to JIVE's dedicated program variable, $\mathit{res}V$.

- **\typeof** returns the most specific dynamic type of an expression. In JIVE, function *typeof* is defined for this purpose.

- `<:` denotes the (reflexive) subtype relation. In JIVE, function $\preceq$ is defined for this purpose.

- **\type** is used to introduce type literals into expression contexts. JIVE has three different functions for this purpose: *at*, *ct*, and *it*. Their usage depends on whether the type given as parameter is an abstract, concrete or interface type, respectively. For instance the JML expressions

  \typeof(myObj) `<:` **\type**(Account)

  would be translated to

  $$\mathit{typeof}(\mathit{myObj}) \preceq \mathit{ct}(\mathit{Account})$$

  in case type Account is a concrete type.

- JML allows one to use the "*" wildcard character to refer to all fields of an object (e.g. "o.*") or all elements of an array (e.g. "arr[*]"). It is mostly used in assignable clauses.

  JIVE currently does not support this, mostly because (at the time of writing) the JML compiler itself does not fully support its usage.

4.5 Handling of JML statements and annotation statements

JIVE requires user-specified invariants for loops. At the time of writing, users of JIVE are always prompted for a loop invariant even if the JML specification contained a **maintaining** clause. While this is surely inconvenient, it is important to note that loop invariants often contain terms which are not expressible on the JML level. For instance, aliveness information.

JIVE does not support **assert** and **assume** statements, however, they do not pose conceptual difficulties. We are planning to support these statements in the near future.

As mentioned in Section 4.1, **set statements** are simply turned into DJC assignments, since ghost fields are handled as concrete fields.

4.6 Specification library

JIVE supports specification-only types. Such types can have specifications without any provided implementation. This is useful for instance to provide specification for API types which allows one to reason about programs that call methods of the API.

Such specification-only types are placed in `.spec` files in a dedicated directory. JIVE comes with a small set of specified API types, for instance, `Object` and `Exception`. But users can freely add other specification-only types to the directory.

The Hoare-triple generation mechanism for such specification-only types is the same as for other types. However, the generated triples are assumed to hold and thus need not be proven—due to this property we call them *closed triples*. Closed triples can be used for the verification of other triples.

4.7 Universes modifiers

Currently JIVE makes only use of the **rep** modifier and only for the relaxed semantics of assignable clauses. That is, fields declared with the **rep** modifier may be modified even if they are not mentioned in the assignable clause. The exact formalization of this semantics was given in Section 4.3.

Note that the JML compiler had to be slightly modified to enable this relaxed semantics. Otherwise the purity analysis performed by the compiler would complain about undeclared modifications of locations. Furthermore, due to this very limited use of Universe modifiers the compiler need be launched so that it would only parse, but not check them.

For a future version of JIVE we plan to use the full power of the Universe type system to support modular verification based on the *relevant object invariant semantics* [28].

4.8 Features not supported by JIVE

We give a summary of JML Level 0 constructs that are currently not supported by JIVE.

- JIVE currently does not take loop invariants specified on the JML level into account. Loop invariants have to be provided during the use of JIVE when the tool prompts the user for a loop invariant.
- DJC, the input language of JIVE, does not support arrays. Thus, **\nonnullelements** expressions are not supported.
- the translation of **initially** clauses is not implemented.
- the translation of **fresh** expressions is not implemented.
- **\TYPE** type specifications are not implemented.
- the "*" wildcard to refer to all fields of a target object is not translated.
- **assert** and **assume** statements are not implemented.

5 Related work

Although there are many verification tools for object-oriented languages, to our knowledge, the literature contains very few descriptions of translations like ours: from some specification language to the underlying logic of a verification tool. However, this is probably due to the relative simplicity of the translation of the basic specification elements. Descriptions of translations in the literature are related to the LOOP, the KeY, and the Krakatoa tools.

The LOOP compiler [4] translates sequential Java programs and JML specifications into PVS or Isabelle theories using a denotational semantics [16]. The use of denotational semantics makes proof obligations look fundamentally different from ours and we believe less intuitive too.

In case of abrupt termination or non-termination of specification expressions, the compiler renders preconditions to true and postconditions to false [5]. This means that nothing can be assumed and nothing can be proved in pre- and postconditions, respectively. This solution is stricter than ours, which allows one to assume/prove parts of pre- and postconditions that do not terminate abruptly.

Breunesse and Poll [6] describes the handling of model fields in LOOP. They propose two solutions. Their first solution uses existential quantification to ensure that the representation relation of a model field is satisfiable. The second solution transforms model fields into pure methods. This solution requires a sound encoding for methods which Breunesse and Poll do not address.

The KeY tool [1] verifies Java Card code against its OCL (Object Constraint Language, part of the UML standard) specification attached to the corresponding UML class diagram. The translation of OCL

specifications into first-order logic is described in [3]. Besides OCL, KeY supports the verification of JML specifications too. A semi-formal description of their translation is given in [13].

The KRAKATOA tool verifies Java Card programs annotated with JML specifications. It translates the program and its specification to the WHY language [14]. A rudimentary description of this translation is given in [23].

6 Future work

This report described the translation of the basic JML constructs that are currently supported by JIVE. Here we sketch our development plans for later versions of the tool.

Modularity. Currently JML does not support the modular verification of Java programs. For instance, as described in Section 4.2.1, postconditions typically conjoin the predicate *INV*(\$) which means that all object invariants need be proven for every method.

In the future we plan to apply means of modular specification and verification. There are different approaches that could be used:

- Our group developed an ownership programming model and an ownership type system [27] for a subset of Java. Approaches that are based on a type system have the great advantage that they are automatically and mostly statically checkable, but on the other hand are typically not flexible enough to handle real-world programs. Based on such a model, a modular verification technique of invariants is possible [28].
- Other methodologies (e.g. the Boogie methodology [2]) offer more flexibility, but cannot be checked statically and require certain properties to be verified. Still they have the advantage that proof obligations are usually much simpler than what classical techniques yield.

It requires future research to see what approach or what combinations of different approaches fit the JIVE tool best.

Model classes. The JML distribution contains a library of model classes. These classes provide mathematical models for data types such as sequences or sets and are used for specification-only purposes. Currently our handling of model classes relies on their specifications. This approach might be unsound if the specifications are not consistent and may lead to unprovable proof obligations, for instance, if model methods are specified in a mutually recursive way.

Thus, we plan to translate the JML library into a program-independent Isabelle theory containing abstract data types. In the first step, the translation would still be based on the specification of model classes, thus, well-definedness would still not be ensured. However, in a next step, users could refine the data types by eliminating ill-defined specifications. A first step of inspecting the feasibility of this approach is presented in [25].

Visibility of specifications. As mentioned in Section 4.2.1, visibility of invariants is not taken into account, for instance, private invariants are also conjoined to *INV*(*OS*). Thus, a private invariant of some type C is assumed to hold in pre-states and is to be proven to hold in post-states for methods residing in types other than C too. A solution we might want to consider is the splitting of the program invariant into several formulas representing the invariant of different visibility levels and contexts as proposed in [26, Section 3.1.1.1].

One source of unsoundness of the tool is that one might use the private specifications of a method m in type C when proving the method body of a method n in a type other than C. In the current implementation this breach of visibility is only present in case m is a static method or the call of m is a super-call.
We do not know of any other tool that handles these cases correctly.

Wider JML support. We are planing to support a wider range of JML constructs in future versions of JIVE. In particular, we are interested in the support of method calls in specifications [9] and the full integration of the Universe type system [11].

Acknowledgments

We thank Werner Dietl, Jenny Jin, Hermann Lehner, and Nicole Rauch for their comments and suggestions on earlier drafts of this report. We are grateful to Ghislain Fourny who implemented a large part of the translation in Jive.

References

[1] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and System Modeling*, 2004.

[2] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[3] B. Beckert, U. Keller, and P. H. Schmitt. Translating the Object Constraint Language into first-order predicate logic. In *Proceedings, VERIFY, Workshop at Federated Logic Conferences (FLoC), Copenhagen, Denmark*, 2002.

[4] J. v. d. Berg and B. Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Software (TACAS)*, number 2031 in Lecture Notes in Computer Science, pages 299–312. Springer-Verlag, 2001.

[5] J. v. d. Berg, E. Poll, and B. Jacobs. First steps in formalising JML: exceptions in predicates. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs. Proceedings of the ECOOP'00 Workshop*. Fernuniversität Hagen, 2000.

[6] C.-B. Breunesse and E. Poll. Verifying JML specifications with model fields. In *Formal Techniques for Java Programs. Proceedings of the ECOOP'2003 Workshop*, 2003.

[7] Y. Cheon. *A Runtime Assertion Checker for the Java Modeling Language*. PhD thesis, Iowa State University, April 2003.

[8] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 33(10) of *ACM SIGPLAN Notices*, October 1998.

[9] A. Darvas and P. Müller. Reasoning About Method Calls in Interface Specifications. *Journal of Object Technology (JOT)*, 5(5):59–85, June 2006.

[10] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, Systems Research Center, HP Labs, 2003.

[11] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, October 2005.

[12] K. Dräger, M. Gawkowski, and N. Rauch. Jive 2.0, The Java Interactive Verification Environment Theoretical Background Information. Department of Computer Sciences, Technische Universität Kaiserslautern, September 2005.

[13] C. Engel and A. Roth. KeY Quicktour for JML. Universität Karlsruhe.

[14] J.-C. Filliâtre. Why: a multi-language multi-prover verification condition generator. Technical report, Université Paris-Sud, France, 2003.

[15] D. Gries and F. B. Schneider. Avoiding the undefined by underspecification. In J. van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 366–373. Springer Verlag, 1995.

[16] B. Jacobs and E. Poll. A logic for the Java Modeling Language JML. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering (FASE)*, number 2029 in Lecture Notes in Computer Science, pages 284–299. Springer-Verlag, 2001.

[17] G. Leavens, A. Baker, and C. Ruby. Preliminary design of JML: A notation for detailed design. Technical report, Iowa State University, Last revised June 2004.

[18] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. *JML Reference Manual*. Iowa State University, Last revised October 2004. Available from http://www.jmlspecs.org.

[19] G. T. Leavens and J. M. Wing. Protective interface specifications. *Formal Aspects of Computing*, 10:59–75, 1998.

[20] K. R. M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995.

[21] K. R. M. Leino. Data groups: Specifying the modification of extended state. In *Proceedings of the 1998 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.

[22] K. R. M. Leino and P. Müller. A verification methodology for model fields. In P. Sestoft, editor, *European Symposium on Programming (ESOP)*, volume 3924 of *Lecture Notes in Computer Science*, pages 115–130. Springer-Verlag, 2006.

[23] C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa Tool for Certification of Java/JavaCard Programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58:89–106, 2004.

[24] J. Meyer, P. Müller, and A. Poetzsch-Heffter. The JIVE system—implementation description. 2000.

[25] M. Miragliotta. Specification model library for the interactive program prover JIVE. ETH Zurich, 2004.

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

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

[28] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. Technical Report 424, Department of Computer Science, ETH Zurich, 2004.

[29] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[30] A. Poetzsch-Heffter and P. Müller. Logical foundations for typed object-oriented languages. In D. Gries and W. De Roever, editors, *Programming Concepts and Methods (PROCOMET)*, 1998.

[31] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *European Symposium on Programming (ESOP)*, volume 1576, pages 162–176. Springer-Verlag, 1999.

[32] A. D. Raghavan and G. T. Leavens. Desugaring JML method specifications. Technical Report 00-03d, Iowa State University, Department of Computer Science, July 2003.

[33] N. Rauch and B. Wolff. Formalizing Java's two's-complement integral type in Isabelle/HOL. In T. Arts and W. Fokkink, editors, *Proc. 8th International Workshop on Formal Methods for Industrial Critical Systems*, volume 80 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 40–56, Røros, Norway, June 2003. Elsevier.

[34] N. Rauch and B. Wolff. Formalizing Java's two's-complement integral type in isabelle/hol. Technical Report 458, ETH Zürich, 11 2004.