Total Correctness of Recursive Functions using JML4 FSPV

George Karabotsos, Patrice Chalin, Perry R. James, Leveda Giannas
Dependable Software Research Group,
Dept. of Computer Science and Software Engineering,
Concordia University, Montréal, Canada
{g_karab,chalin,perry,leveda}@dsrg.org

ABSTRACT
JML4 is a next generation tooling and research platform for JML. JML4, currently in development, aims to support the integrated capabilities of Runtime Assertion Checking (RAC), Extended Static Checking (ESC), and Full Static Program Verification (FSPV). In this paper, we present the JML4 FSPV Theory Generator (TG) that aims to study the adequacy of Isabelle/Simpl as the underlying verification condition language. In particular we study Isabelle/Simpl with respect to proving total correctness of recursive programs. Simpl is a Hoare-based logic for a sequential imperative programming language along with a verification system. It is written in Isabelle/HOL and has been proven sound and relative complete.

Categories and Subject Descriptors
D.2.4 [**Software Engineering**]: Software/Program Verification—*Programming by contract, Correctness proofs*; F.3.1 [**Logics and Meaning of Programs**]: Specifying and Verifying and Reasoning about Programs—Mechanical verification.

General Terms
Reliability, Languages, Theory, Verification.

Keywords
Java, Java Modeling Language, Full Static Program Verification.

1. INTRODUCTION
The Java Modeling Language (JML) is a Behavioral Interface Specification Language (BISL) for Java [14]. A number of tools exist that recognize JML annotated Java programs and can help in demonstrating their correctness [4]. These tools perform verification using one or more of three main verification methods: Runtime Assertion Checking (RAC) [7], Extended Static Checking (ESC) [8], and Full Static Program Verification (FSPV) [12].

While RAC and ESC are fully automated and generally easy to use, these verification techniques are either unsound and/or incomplete by nature of the technique. Unfortunately, this is unacceptable for safety and security critical applications (e.g. SmartCard applications such as electronic purses used in commercial transactions and medicare cards used to hold vital patient information) for which soundness and completeness are vital. FSPV, on the other hand, has the potential to be both sound and complete. In this paper, we present the *FSPV Theory Generator (TG)*, the FSPV component of JML4—a next generation tooling and research platform for JML. In particular, we present initial results with respect to proving the total correctness of recursive functions. To our knowledge, the JML4 FSPV TG is the *first*:

- JML tool to enable the total correctness of recursive functions to be proven, such as the one shown for Factorial in Figure 1, and
- FSPV tool to be based on an underlying theory that has been proven sound and complete, and this within a mechanical theorem prover.

Creation of the FSPV TG is also timely, since neither of the two "first generation" FSPV tools (JACK, LOOP) is still being actively maintained.

We present:

- The *translation* process used to generate Isabelle/Simpl [20] theories from Java programs.
- Our *experience* in generating and proving Simpl theory Verification Condition (VC) lemmas for JML annotated Java programs.

Isabelle/Simpl is a theory built atop Isabelle/HOL for an IMP-like [22] sequential imperative programming language with loops and procedures supported by specification constructs (e.g., via pre- and post-conditions).

The rest of the paper is structured as follows. In the next section, we describe Isabelle, Simpl, and JML4. Section 3 presents the FSPV TG followed by an account of its use and subsequent verification of its generated theories in Section 4. In Section 5 we present related work. Finally conclusions and future work are given in Section 6.

2. BACKGROUND

2.1 Isabelle
Isabelle [18] is a theorem proving framework. It provides the necessary proving apparatus to define new logics. This machinery includes Isabelle's meta-logic (Isabelle/Pure), the classical reasoner, and the simplifier. Additionally, existing logics can be extended, thus defining new ones. Newly constructed object logics can be further enhanced with new syntax by making use of Isabelle's syntax transformations. These transformations can be specified using relatively simple rules defined within the theory or

```
public class Factorial {
  //@ requires n >= 0;
  //@ ensures  \result ==
  //@   (\product int j; 1 <= j && j <= n ; j);
  //@ measured_by n;
  public static int fac(final int n) {
    if(n == 0)
      return 1;
    else
      return n * fac(n-1);
  }
}
```

Figure 1: Recursive factorial method

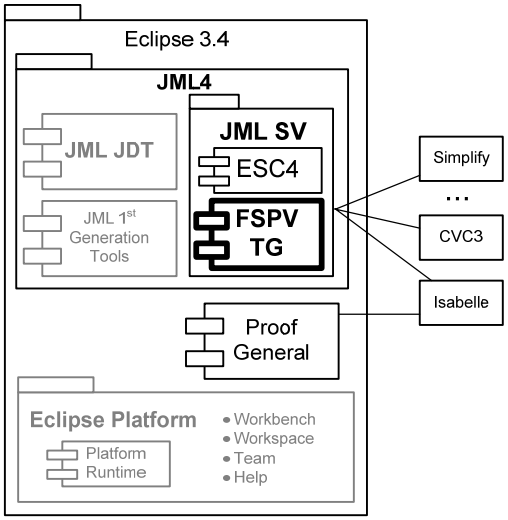

Figure 2: JML4 component diagram

with more complex but more powerful translation functions coded in ML.

Isabelle/HOL, a realization of High Order Logic for Isabelle, is just one of these logics defined atop of Isabelle/Pure. It is the most complete of all of the object logics written for Isabelle so far. This reason, among others, is why Isabelle/HOL has served as the basis for a number of additional logics. Some of these include the Logic for Computable Functions (Isabelle/HOLCF), and logics for sequential imperative programs with Hoare semantics defined such as Bali [17] and Simpl.

2.2 Simpl

Simpl [7] is a theory written and *proven* sound and complete in Isabelle/HOL for a generic sequential imperative programming language. The Simpl theory includes definitions of syntax, big- and small-step operational semantics, a set of Hoare rules both for partial and total correctness, and weakest-precondition semantics (via the `vcg` and `vcg-step` proof methods) [9]. It is expressive enough for many language constructs that exist in modern programming languages. These include: global and local variables, exceptions, abnormal termination, breaks out of loops, procedures, as well as expressions with side-effects. Simpl also has theories for reasoning about the heap and references, thus allowing for the expression of linked data structures.

Essential elements of a typical Simpl theory include states, procedure declarations, and Hoare triples. The state takes the form of a `hoarestate` statement, which contains the list of variables used in the Hoare triple—examples will be given further below. Procedures are declared using Simpl's `procedures` declaration and have the following form:

```
procedures
  N (x::τ¹, y::τ², …| z::τ³)
  where v::τ⁴ … in B
```

where `N` is the procedure's name, `x` and `y` the formal parameters, τ^n a type, `z` the return value, `v` a local variable, and `B` the body. A `procedures` declaration is syntactic sugar for a number of deductive elements that are dynamically generated by Simpl and include a `locale`[1] and a `hoarestate`. All such locales are named using the name of the procedure and the prefix `_impl`.

Hoare triples have the usual form and in Simpl are written as:

```
Γ, Θ ⊢ {|P|} B {|Q|}, {|R|}
Γ, Θ ⊢t {|P|} B {|Q|}, {|R|}
```

for partial and total correctness, respectively. `Γ` is the procedure environment, `Θ` is a set of Hoare rules used as assumptions, `P` is the precondition, `B` is the body, and `Q` and `R` are the postconditions for normal and abrupt termination, respectively.

2.3 JML4

JML4 [5] is a next generation research platform for JML. It is an Eclipse-based Integrated Development and Verification Environment (IVE)—see Figure 2. Users can write their Java programs, annotate them with JML specifications, and prove them correct within the same environment using RAC, ESC, or FSPV.

Currently, JML4 supports JML's non-null type system (both statically and at runtime), the ability to read and make use of the extensive JML API library specifications, and basic RAC. Our research group, in addition to contributing to the basic infrastructure of JML4, is focusing on a new static verification component called the JML Static Verifier (SV). The JML SV offers support for ESC and FSPV. We examine the FSPV component in more detail in the sections that follow.

3. JML4 FSPV THEORY GENERATOR

In this section we present FSPV TG. Central to FSPV TG is a translator that takes Java programs along with their associated JML specifications and generates one or more Simpl theory files.

The choice of Simpl as a target VC language for our FSPV tool is motivated by two main reasons. Firstly, the generation of the VC is fully captured within Simpl, which as mentioned above, has been proven sound and complete. The alternative (and the norm) is to programmatically define VC generation and in some cases prove soundness, most of the time this is done by hand. Secondly, Simpl's syntax is such that rather than expressing lemmas as "low-level" VCs, we express them directly as Hoare triples.

At its current level, the FSPV TG supports a handful of JML and Java language elements, including method calls. Type-wise, only Integers and Booleans are supported while initial support for class related elements such as fields and methods are in place. A functional set of Java statements and expressions are supported. These include local-variable declarations with initialization and conditional and while-loop statements. Most arithmetic, relational, and logical operators are supported, including those with side-effects. Lightweight JML contracts and loop annotations are supported. All these elements are translated into Isabelle/Simpl using FSPV TG's translator.

FSPV TG's current translation phases, along with their individual inputs and outputs, can be seen in Figure 3. The first phase is named *TheoryTranslation*. The input to the first phase is the *JML+Java Abstract Syntax Tree* (AST) for a compilation unit. A compilation unit contains AST nodes for type declarations, which in turn contain type member nodes such as method declarations, fields, etc. The result of this phase is a (generic) *Theory AST* (Figure 4). This resulting theory AST consists of both a list of variables containing field-related information and one or more lemma AST nodes. Each lemma node is a translation of a single Java method declaration and represents the proof obligation for that method. Proof of the lemma establishes the correctness of the method with respect to its specification. A lemma node is a pair of:

[1] A locale is Isabelle's construct for parameterized theories

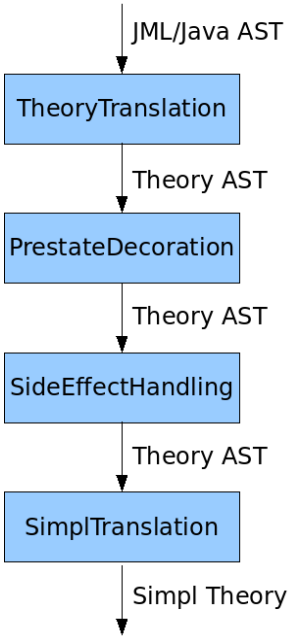

Figure 3: FSPV TG Phases

- a variable list containing all parameters and local variables declared in the method
- a Hoare triple containing the translations of the JML pre- and post-conditions, as well as the translation of the method body.

In the next phase, *PrestateDecoration*, the Theory AST is decorated with pre-state information. This entails storing the pre-state of the method parameters (since they can be modified within the method body) as well as the handling `\old` JML expressions. The result of this phase is an enriched Theory AST with additional variables, assignment nodes, and simplified `\old` JML expressions.

Additionally, in this stage we perform data analysis of the code in the presence of while loops. Translating while loops requires some care. Simpl adopts the classic Hoare rule for while loops whereas in JML, the assumption is that only a while body's assignment targets are "havocked"[2]—all other variables are assumed to remain unchanged. As such, the loop invariant is augmented to maintain additional state information—i.e., constraints that the non-havocked variables remain unchanged. Examples of this are presented in Section 4.

The third phase is called the *SideEffectHandling*. This phase translates expressions with side effects into a more palatable form, based on examples from the Simpl distribution of simplifying such expressions. To allow for this translation we introduce additional variables and assignment statements that hold intermediary results. To illustrate this, consider the following Java statement containing an expression with side-effects:

```
a *= b - i++;                                    (1)
```

It is translated into the following sequence of Java statements:

```
a0 = a;
i0 = i;
i = i + 1;
a = a0 * (b - i0);
```

This will translate into Isabelle using Simpl's notion of a *binder variable*: the expression $E' >> v \ . \ E(v)$ evaluates to $E(v)$ in which v, if it occurs free, will have the value E', i.e. $E(E')$. The Simpl translation of (1) is:

```
a >> a0.
  i >> i0.
    i :== i + 1 ;;
    a :== a0 * (b - i0)
```

The last phase, called *SimplTranslation*, is responsible for generating the Simpl theory. For each theory AST, an Isabelle theory is created containing a `hoarestate` block for static and instance fields. For each lemma Theory AST node, a Simpl procedure and an Isabelle `lemma` block statement is created. The procedure contains the translation of the Java method into Simpl, while the lemma is there to prove the method correct with respect to its specification.
Examples will be given in Section 4.

Variables	$y : V$
Integers	$n : Z$
Boolean	$b : \{T,F\}$
Operators	op ::== + | - | * | / | ∨ | ∧ | = | != | ++ | -- | += | -= | *= | /= | :=
Expressions	e ::== $y \mid n \mid b \mid e\ op\ e \mid op\ e \mid e\ op$
Statements	s ::== y := e | **WHILE** e **INV** e **VAR** e s | **IF** e **THEN** s **ELSE** s | s ; s
Types	$\tau : \Gamma$
Lemma	l ::== $(y :: \tau)^*$ $\{e\}\ s\ \{e\}$
Theory	t ::== $(y :: \tau)^*$ l^*

Figure 4: Theory language abstract syntax

4. FSPV BY EXAMPLE

In this section, we present examples of recursive functions specified in JML and proven using Isabelle/Simpl. Each example allows us to highlight a particular capability of the JML4 FSPV TG or limitations in JML with respect to its linguistic ability to support the specification of recursive functions, especially for the purpose of proving total correctness. Note that for these examples, the resulting Simpl theories have a close resemblance to their associated Java classes. We found this quite pleasing since source code parts are easily identifiable in the corresponding theory.

4.1 Factorial

In Figure 1, presented earlier, we define the `Factorial` class with a single recursive method name `fac`, which returns the factorial of its integer argument. Our aim is to prove the method correct and that it terminates. The JML `measured_by` clause allows us to provide a measure that we can use to prove termination. A measure is a well-founded relation from a function to the natural numbers. Termination is achieved when the arguments of each recursive method call decrease with respect to the measure. While the definition of the `fac` method is simple, we note that it is already beyond the capabilities of ESC/Java2 due to the use of a generalized numeric quantifier in the method contract. Hence, factorial allows us to demonstrate the use, translation, and verification of JML's generalized numeric quantifiers such as `\product`. Moreover, ESC/Java2 does not support the `measured_by` JML statement. The corresponding

[2] Nothing can be assumed about the value of havocked variables.

```
theory Factorial imports Vcg SetHelper begin
  procedures
    Factorial_fac_int(n::int|result'::int)
  "IF ´n = 0
   THEN
     ´result' :== 1
   ELSE
     CALL Factorial_fac_int(´n - 1) ≫ n1.
     ´result' :== ´n * n1
   FI"
  lemma (in Factorial_fac_int_impl) Factorial_fac_int_spec:
  "∀n σ.
     Γ ⊢\<^sub>t {|σ. n = ´n ∧ ´n ≥ 0|}
     ´result' :== PROC Factorial_fac_int(´n)
     {| ´result'=(∏ {j :: int. ((1≤j) ∧ (j≤n))}) |}"
    apply(hoare_rule HoareTotal.ProcRec1
          [where r="measure (λ (s,p). nat \<^bsup>s\<^esup>n )"])
    apply(vcg)
    apply(auto)
  done
end
```

Figure 5: Simpl Theory for Factorial

Simpl theory is generated as part of the compilation process when the user selects the appropriate JML4 compiler options. The theory generated for Factorial is given in Figure 5.

The theory has two main parts: a Simpl `procedures` and an Isabelle `lemma` declaration. If more methods had been present in the Java class declaration then additional pairs of `procedures` and `lemma` declarations would have been given, one for each method. The `procedures` declaration contains the translation of the Java method in Simpl as well as all variables referenced by the program including `` `result' ``, a special variable added by the FSPV TG to hold the return value. The name of the class, the name of the method, and the method's signature are used to name the corresponding Simpl procedure. Encountering this procedure declaration, Simpl dynamically generates the `Factorial_fac_int_impl` locale that contains all the deductive machinery required for reasoning about the procedure. This locale is subsequently used in the `lemma` block to prove the procedure correct with respect to its specification.

We can identify the lemma definition enclosed within quotes. This definition follows the general format of a Simpl lemma definition proving total correctness (see Section 2.2)[3]. The lemma definition contains the Hoare triple to be proven, followed by its proof. We can clearly identify the pre- and post-condition at the top and bottom of the lemma enclosed within `{|` and `|}` character sequences which are used to denote assertions. Additionally, we bind the value of the input parameter to the logical variable `n` which is used in the postcondition in order to preserve the pre-state value of `´n`. The logical variable `σ` represents the pre-state; `σ` is always generated though it is not used in the examples presented here. In between, is a call to the `Factorial_fac_int` procedure. It is worth noting how JML `\product` quantified expressions are translated to Isabelle/HOL's product definition `∏` using an Isabelle set comprehension to specify the range. Isabelle/HOL's set theory is typed and extensive. It allows for set comprehensions and ranges which are ideal when translating JML numeric quantifiers.

To prove this procedure correct and that it terminates we need to provide a well-founded relation and to prove that subsequent recursive calls are decreasing with respect to its arguments—for our factorial example this means that subsequent recursive calls

[3] The \<^sub>t is how ProofGeneral subscript characters. Unfortunately not all of Proof General's X-symbols are supported in the Eclipse plug-in.

```
public class McCarthy {
  //@ requires n >= 0;
  //@ ensures \result == (100 < n ? n-10 : 91);
  //@ measured_by 101 - n;
  public static int f91(int n) {
    if(100 < n)
      return n - 10;
    else
      return f91(f91(n + 11));
  }
}
```

Figure 6: Recursive McCarthy's 91 Method

```
theory McCarthy imports Vcg begin
  procedures
    McCarthy_f91_int(n::int | result'::int)
  "IF 100 < ´n
   THEN
     ´result' :== ´n - 10
   ELSE
     CALL McCarthy_f91_int(´n + 11) ≫ n1.
       ´result' :== CALL McCarthy_f91_int(n1)
   FI"
  lemma (in McCarthy_f91_int_impl) McCarthy_f91_int_spec:
    "∀n σ. Γ ⊢\<^sub>t
       {|σ. n=´n ∧ ´n≥0|}
       ´result' :== PROC McCarthy_f91_int(´n)
       {| ´result'=(if 100 < n  then  n - 10 else 91 ) |}"
   apply(hoare_rule HoareTotal.ProcRec1
    [where r="measure (λ (s,p). nat (101 - \<^bsup>s\<^esup>n) )"])
   apply(vcg)
   apply(auto)
  done
end
```

Figure 7: Simpl Theory for McCarthy's 91 Function

are made using smaller non-negative integer values. Isabelle/HOL provides us with such a mechanism via the `measure` clause. The measure clause for this particular example is just the input parameter and it has the following form: `measure λ(s,p). nat ˢn`. To introduce this measure to our proof we make use of the `HoareTotal.ProcRec1` rule and we instantiate the `?r` schematic [18] variable with the measure using the `where` theorem modifier.

To complete this proof we need to provide additional properties pertinent to the set comprehensions used in the post-condition. These are included as simplification rules in the `SetHelper` theory (imported by the `theory` statement) which is provided in Appendix A. Finally, we complete the proof using two applications of the `vcg` and `auto` methods.

To work with the theory we use Eclipse's ProofGeneral plug-in [1] which is a generic front-end for interactive theorem provers supporting Isabelle. It is through Proof General that we prove this theory correct following the proof steps described in the previous paragraphs.

4.2 McCarthy's 91 Function

Our next example contains an implementation of McCarthy's 91 function [15]. The `f91` method, seen in Figure 6, is defined over positive integers and returns 91 for all `n <= 100` otherwise it returns `n - 10`. The measure for the function is remarkably simple: `101 - n`. McCarthy's 91 function is interesting because of its use of nested recursion.

The FSPV generated theory is shown in Figure 7. Like in the previous example, a Simpl procedure and its associated Simpl specification lemma are generated. We prove correctness and termination within Eclipse using the associated Proof General plug-in. Despite the nested recursion we are able to verify the procedure correct and that it terminates with relative ease: i.e., by merely asking Simpl to generate the verification condition (vcg),

```
class Fibonacci {
  //@ public static native int fib_spec(int n);

  //@ requires n>=0;
  //@ ensures \result ==  fib_spec(n);
  //@ measured_by n;
  public static /*@ pure */ int fib(int n) {
    if(n == 0)
      return 0;
    else if (n == 1)
      return 1;
    else
      return fib(n-1) + fib(n-2);
  }
}
```

Figure 8: Fibonacci Method (using native `fib_spec()`)

which Isabelle's auto method is then able to discharge without further user intervention. Surprisingly, our proof in Simpl is simpler than the corresponding proof for a native Isabelle/HOL function definition of the 91 function presented in [13].

4.3 Fibonacci Numbers

Our next example is a recursive method that calculates Fibonacci numbers (see Figure 8). The difference with respect to the previous cases is that in this example we make use of the `native` JML feature, recently proposed by Julien Charles [6]. In essence, this feature declares pure JML methods without an explicit definition. The definition is instead provided using the underlying target logic that JML annotated Java code is translated to. This provides for a more natural way of proving recursive methods that have in their specification recursive method calls. Moreover, it allows us to illustrate the definition of Isabelle/HOL functions and their use within Simpl assertions.

Figure 9 presents the generated theory suitably edited to include a definition of `fib_spec()` and our modifications that prove the method correct and that it terminates with respect to its specification and its measure, respectively.

The Simpl procedure declaration of `Fibonacci_fib_int` contains the translation of the Java statements and expressions into Isabelle/Simpl. Notice how binder variables are used to store the intermediate results of the recursive calls.

The `fib_spec()` function is the definition of the corresponding native pure methods. We make use of the Isabelle special polymorphic value `arbitrary` which is used to denote an arbitrary value. This is required because Isabelle/HOL functions are total by definition—i.e. we underspecify the function for negative integers. For every Isabelle/HOL function two proof obligations are required to be satisfied: one for completeness and compatibility of patterns and another for termination [13]. Their respective proofs follow the definition. It is worth mentioning that Isabelle/HOL provides a simpler form of defining functions where both of these proofs are satisfied automatically, however, the default termination proof (based on lexicographic order) is not sufficient for the `fib_spec` function—hence, the use of the "long" form.

The final part of this theory is the specification lemma. The proof proceeds as in the previous cases where the `HoareTotal.ProcRec1` rule is used, instantiated by a well-founded relation (via `measure`) and followed by an application of the `vcg` and `auto` methods.

Supporting reasoning about pure model methods having contracts that fully capture their behavior is possible (see Figure 10). This can be accomplished by using inductive sets to encode

```
theory Fibonacci imports Vcg begin
  procedures
    Fibonacci_fib_int(n::int | result'::int)
  "IF ´n = 0
   THEN
     ´result' :== 0
   ELSE
     IF ´n = 1
     THEN
       ´result' :== 1
     ELSE
       CALL Fibonacci_fib_int(´n - 1) ≫ n1 .
       CALL Fibonacci_fib_int(´n - 2) ≫ n2 .
       ´result' :== n1 + n2
     FI
   FI"

  function fib_spec :: "int ⇒ int" where
  "fib_spec n =
    (if n = 0 then 0 else
      (if n=1 then 1 else
        (if n < 0 then arbitrary
          else (fib_spec (n - 1)) + (fib_spec (n - 2)))))"
  by(pat_completeness, auto)
  termination by (relation "measure (λn. nat n)",auto)

  lemma (in Fibonacci_fib_int_impl) Fibonacci_fib_int_spec:
  "∀n σ. Γ ⊢\<^sub>t
       {|σ. ´n=n ∧ ´n≥0|}
       ´result' :== PROC Fibonacci_fib_int(´n)
       {|´result'=fib_spec(n)|}"
  apply(hoare_rule HoareTotal.ProcRec1
    [where r="measure (λ (s,p). nat \<^bsup>s\<^esup>n )"])
  by(vcg,auto)
end
```

Figure 9: Simpl Theory for Fibonacci

```
class Fibonacci {
  //@ requires n>=0;
  //@ ensures \result == (n==0)? 0 : (n==1) ? 1
  //@  : fib_spec(n-1)+fib_spec(n-2);
  //@ measured_by n;
  //@ public static pure model
  //@                       int fib_spec(int n);

  //@ requires n>=0;
  //@ ensures \result == fib_spec(n);
  //@ measured_by n;
  public static /*@ pure */ int fib(int n) {
    ...
  }
}
```

Figure 10: Fibonacci Method with fib_spec() as a model method

the method contract and then proving that the inductive definition is functional.

4.4 Ackermann's Function

In the previous examples we have dealt with functions having trivial measures. In this section we illustrate a total termination proof for a recursive implementation of the Ackermann function [15] (see Figure 11) which has a non-trivial measure. This measure is a well-founded relation on pairs of non-negative integers. In the process we also recognize the inadequacy of the `measured_by` clause in specifying this measure. Once more we make use of a native pure JML method to specify the post-condition. As we shall see, its definition in Isabelle also helps in making the case of preferring natural numbers instead of integers when working with non-negative values.

The complete theory that includes our modifications is presented in Figure 12. In addition to the `procedures` and `lemma` declarations we have defined two Isabelle/HOL functions

```
public class Ackermann {
  //@ public static native int ack_spec(int n);

  //@ requires n >= 0 && m >= 0 ;
  //@ ensures \result == ack_spec(n,m);
  public static int ack(int n, int m) {
    if(n == 0)
      return m + 1;
    else
      if(m == 0)
        return ack(n-1, m);
      else
        return ack(n-1, ack(n, m-1));
  }
}
```

Figure 11: Ackermann Method

(`ack'` and `ack_spec`) and a `lemma` declaration (`distrib_minus_int`) that proves that Isabelle's `nat` operator distributes over subtraction of integers, where the right hand side of the subtraction is the integer 1.

The `ack_spec` function is implemented over integer values that return the Isabelle `arbitrary` value when either one of its arguments is a non-negative number—in all other cases it makes use of the value returned by the `ack'` function. The `ack'` function is an implementation of the Ackermann function over natural numbers. It is possible to avoid writing the `ack'` function altogether and incorporated the remaining cases in the `ack_spec` definition—in fact our first attempts in a definition of the native method followed this approach. We were successful in completing an integer only definition of `ack_spec`. However, when this is used within the `Ackerman_ack_int_int_spec` lemma the Isabelle simplifier enters what it seems an infinite loop. In general, natural number based definitions are easier to work with in Isabelle/HOL. Hence, by using a natural number implementation of the Ackermann function as a first step we are able to prove the corresponding Simpl procedure correct. We are confident that even with our original approach a proof of correctness is achievable given additional investment on our part.

In the `Ackermann_ack_int_int_spec` lemma we have manually inserted the measure using the `HoareTotal.ProcRec1` rule as to demonstrate that Isabelle/Simpl is capable of proving termination of the Ackermann function. The measure we provide is in fact a list of two measures. As such they do not correspond to the current syntax and semantics of the `measured_by` clause. In Isabelle/Simpl such measure lists are specified using the `measures` combinator. This `measures` combinator is a generalization of the measure clause and it constructs a well-founded relation from a list of measures—it is explained in detail in [3]. We continue the proof with a set or repeated applications of the `auto` and `vcg` methods. These methods generate subgoals that each is resolved by cases on the `nat` type followed by an extra application of the `auto` method.

Table 1: A Comparison on Java's FSPV Tools

	LOOP	JACK	Krakatoa Why	FSPV TG Simpl
Maintained	×	×	✓	✓
Open Source	×	✓	✓	✓
Proven Sound	✓	×	✓	✓[1]
Proven Complete	×	×	×	✓[1]
Above two proofs done	in PVS	N/A	by hand	in Isabelle
VC generation done in prover	×	×	×	✓
Termination of recursive functions	×	×	×[2]	✓

[1] Simpl is proven sound and complete. The translation to Simpl is not.
[2] See main text for a qualification of this mark.

```
theory Ackermann imports Vcg begin
  procedures
    Ackermann_ack_int_int(n::int, m::int|result'::int)
  "IF ´n = 0 THEN
     ´result' :== ´m + 1
   ELSE IF ´m = 0 THEN
     ´result' :== CALL Ackermann_ack_int_int(´n - 1, 1)
   ELSE
     CALL Ackermann_ack_int_int(´n, ´m - 1) ≫ m1.
     ´result' :== CALL Ackermann_ack_int_int(´n - 1, m1)
   FI FI"
  function ack' :: "nat ⇒ nat ⇒ nat" where
    "ack' 0 m = Suc m"
  | "ack' (Suc n) 0 = ack' n 1"
  | "ack' (Suc n) (Suc m) = ack' n (ack' (Suc n) m)"
  by (pat_completeness,auto)
  termination
    by(relation "measures [λ(n,m). n, λ(n,m). m]",auto)
  fun ack_spec :: "int ⇒ int ⇒ int" where
    "ack_spec n m =
      (if n<0 then arbitrary else
       (if m<0 then arbitrary else
         int (ack' (nat n) (nat m))))"
  lemma nat_distrib_minus_int [simp]: "∀x. nat (x - 1) = (nat x) - (nat 1)"
   by (auto)
  lemma (in Ackermann_ack_int_int_impl) Ackermann_ack_int_int_spec:
    "∀n m σ. Γ ⊢\<^sub>t
       {|σ. n=´n ∧ ´n≥0 ∧ m=´m ∧ ´m≥0 |}
       ´result' :== PROC Ackermann_ack_int_int(´n, ´m)
       {|´result' =  (ack_spec n m) |}"
   apply(hoare_rule HoareTotal.ProcRec1
      [where r="measures [λ(s,p). nat \<^bsup>s\<^esup>n,
                          \<lambda>(s,p). nat \<^bsup>s\<^esup>m]"] )
   apply((auto|vcg)+,case_tac "nat n",auto,case_tac "nat n",auto)
   by (case_tac "nat m",auto)
  end
```

Figure 12: Ackermann Theory

5. RELATED WORK

In this section we examine three existing FSPV tools.

LOOP. The LOOP tool [12,21] was developed at the University of Nijmegen in Netherlands. LOOP covers a functional subset of sequential Java. In particular, LOOP can handle all of Java Card. Thus, LOOP is able to reason about expressions with side effects, exceptions, inheritance, and overloading. To our knowledge only multi-threading, inner classes and termination of recursive programs are left out.

The LOOP tool is a compiler. Its input is JML-annotated Java source code and its output is theories for the PVS theorem prover. These theories, along with a set of theories named "the prelude," are used as input to the PVS theorem prover when a developer wishes to conduct a verification session. The prelude contains the semantics of both JML and Java. Through user interaction, properties of these JML/Java sources can then be verified. A user working with LOOP-generated theories has a choice between a Hoare logic and two weakest-precondition calculi.

As compared to Isabelle/Simpl, LOOP's Hoare logic has been proven sound using PVS, but not proven complete. To our knowledge, the LOOP tool does not support termination of recursive programs. LOOP incorporates the semantics of JML and Java in its compiler generating primitive formulas which are then used as input to the PVS prover. FSPV-TG, on the other hand, generates Simpl theories which incorporate the semantics of sequential programming languages in terms of Hoare logic and weakest precondition semantics—i.e. the transformation from a Hoare triplet to a primitive formula is done within the prover.

JACK. The Java Applet Correctness Kit (JACK) tool [2] is an Eclipse plug-in. Like LOOP, JACK also translates Java programs into one or more theory files. However, JACK generates theories in a Java-like language called *Java Proof Obligation* (JPO) language. These obligations are generated using weakest precondition semantics which, to our knowledge, has yet to be proven sound. JACK provides support for a number of theorem provers, namely Coq, PVS, B, and Simplify—with Coq and Simplify being the most fully supported. Prover-specific theories are translated using the JPO theories as input. Additionally, JACK supports specification and verification at the bytecode level. Bytecode verification also makes use of a weakest-precondition semantics. In this case, this semantics is proven sound using the pen and paper approach [19].

The differences between the underlying logics of JACK and FSPV TG are similar to those of LOOP. JACK generates primitive formulas in Java, while we make use of Simpl's Hoare rules and weakest precondition semantics to generate the primitive formulas. Additionally, JACK does not support termination proofs for recursive functions.

Krakatoa. Krakatoa is an FSPV tool for JML annotated Java classes. Originally designed to generate theories for the Coq theorem proven it has recently been modified to output programs for the Why tool as well [11].

Why is a multi-tool Verification Condition (VC) generator. The input of Why is a Why program. A Why program may contain assignment, loop, and conditional statements, as well as function declarations. Additionally, it supports throwing and catching exceptions and has limited support for expressions with side-effects. It supports annotations for function declarations and loop statements.

The Why tool transforms input programs into VCs using a weakest-precondition semantics proven sound using the pen and paper approach [10]. The output is one or more theories for a number of provers. These include the automated Yices, CVC3, and the Interactive Coq, Isabelle, and PVS. It is worth noting that Why is general enough that it is used by Caduceus—a front-end for verifying C programs.

Krakatoa is similar to FSPV TG in the sense that it translates Java programs into an intermediate program. However, Why programs are translated into a prover-specific theory using the Why compiler written in Objective CAML. Consequently, it suffers from the same issues as LOOP and JACK with respect to having VCs generated programmatically. Krakatoa does not support reasoning about the termination of recursive methods as indicated by [16]. Nonetheless its underlying intermediate language, Why, does have support for specifying recursive functions (via the `rec` keyword) with measures (via the `variant` keyword).

Table 1 presents a comparison in terms of the soundness and completeness of the underlying logical foundations of these FSPV tools along with our own FSPV TG. Additionally, we report (second to last row) on which tools programmatically generate VCs and which generate them through a theorem prover. Finally, in the last row, we report on tool support for proving termination of recursive programs.

6. CONCLUSION AND FUTURE WORK

We have presented initial work we have done in implementing an FSPV tool in JML4. This FSPV tool makes use of Simpl—a logic for expressing and verifying sequential imperative programs developed within Isabelle/HOL. Simpl's Hoare logic has been proven sound and complete with respect to the programming language semantics. We have illustrated the current level of support that the FSPV TG provides and presented a sample of our experimental test cases. We have focused our attention on proving recursive programs correct and that they terminate.

We have shown programs implementing Factorial and McCarthy's 91 function and how the FSPV TG, at its current state, can correctly prove total correctness. We examined more complicated cases such as Fibonacci and the Ackermann function. In there we employed the recently introduced native feature that allows separating declaration and definition of JML pure methods. This separation allowed for an "easier", a more natural, and a flexible definition of the pure method in the underlying logic. Moreover, we have exposed inadequacies of JML in specifying complex measures such as the one for the Ackermann function.

Through our experiments we believe that we have demonstrated the feasibility of Isabelle/Simpl as a backend proving apparatus for our FSPV TG tool proving recursive programs correct and that they terminate. To our knowledge FSPV TG is unique with respect to applying Hoare logic rules and weakest precondition semantics within an interactive theorem prover.

We reviewed a number of related FSPV tools and we have seen that Simpl is the only logic proven both sound and complete within an interactive theorem prover. Additionally none of our reviewed tools supports total correctness of recursive programs.

We have plans for a number of future additions to this tool. A short-term goal is to make progress towards using pure model methods rather than native methods to specify recursive functions like the one given in our Fibonacci example. We will also be exploring extensions to the `measured_by` syntax of JML so that measures for Ackermann's function can be defined within JML directly.

REFERENCES

[1] Aspinall, D. et al. 2006. Proof general in Eclipse: system and architecture overview. ACM, 45-49.

[2] Barthe, G. et al. 2007. JACK - A Tool for Validation of Security and Behaviour of Java Applications. *In 5th International Symposium on Formal Methods for Components and Objects (FMCO),* , 152-174.

[3] Bulwahn, L. et al. 2007. Finding Lexicographic Orders for Termination Proofs in Isabelle/HOL. In *Theorem Proving in Higher Order Logics*. 38-53.

[4] Burdy, L. et al. 2005. An overview of JML tools and applications. *Int. J. Softw. Tools Technol. Transf. 7*, 3, 212-232.

[5] Chalin, P. et al. 2008. JML4: Towards an Industrial Grade IVE for Java and Next Generation Research Platform for JML. In *Verified Software: Theories, Tools, Experiments*. 70-83.

[6] Charles, J. Adding native specifications to JML. *Formal Techniques for Java-like Programs,* , 2006.

[7] Cheon, Y. and Leavens, G.T. 2002. A runtime assertion checker for the Java Modeling Language (JML). In *International Conference on Software Engineering Research and Practice (SERP '02)*. CSREA Press, Las Vegas, Nevada, 322--328.

[8] Cok, D.R. and Kiniry, J.R. 2004. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an Internet voting tally system. *In Construction and Analysis of Safe, Secure*

and Interoperable Smart Devices: International Workshop, CASSIS 2004 3362, , 108--128.

[9] Dijkstra, E.W. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM 18*, 8, 453-457.

[10] Filliâtre, J. 2003. Verification of non-functional programs using interpretations in type theory. *J. Funct. Program. 13*, 4, 709-745.

[11] Filliâtre, J. and Marché, C. 2007. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In *Computer Aided Verification*. 173-177.

[12] Jacobs, B. and Poll, E. 2004. Java Program Verification at Nijmegen: Developments and Perspective. In *Software Security - Theories and Systems*. 134-153.

[13] Krauss, A. 2008. Defining Recursive Functions in Isabelle/HOL. http://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle/doc/functions.pdf.

[14] Leavens, G.T. 2008. The Java Modeling Language (JML). http://www.eecs.ucf.edu/~leavens/JML/.

[15] Manna, Z. 1974. *Mathematical Theory of Computation*. Mcgraw-Hill College.

[16] March, C. et al. The Krakatoa Tool for Certication of Java/JavaCard Programs annotated in JML. .

[17] Nipkow, T. 2008. Project Bali. http://isabelle.in.tum.de/bali/.

[18] Nipkow, T. et al. 2002. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer.

[19] Pavlova, M. 2007. *Java bytecode verification and its applications*. Ecole Superieure en Sciences Informatiques de Sophia Antipolis.

[20] Schirmer, N. 2005. A Verification Environment for Sequential Imperative Programs in Isabelle/HOL. In *Logic for Programming, Artificial Intelligence, and Reasoning*. 398-414.

[21] Van Den Berg, J. and Jacobs, B. 2001. The LOOP compiler for Java and JML. *Tools and Algorithms for the Construction and Analysis of Systems, number 2031 in Lect. Notes Comp. Sci*, , 299--312.

[22] Winskel, G. 1993. *The formal semantics of programming languages: an introduction*. MIT Press.

APPENDIX A

```
theory SetHelper imports Main begin
 lemma preToInv [simp]: "{x. 1 ≤ x ∧ x ≤ (0::int)} = {}"
 proof -
   show ?thesis by auto
 qed

 lemma setinterval_iff: "{x. a ≤ x ∧ x ≤ b} = {a .. b}"
  by auto

 lemma prodEqProdTimes:
  assumes n: "m - 1 < (n::int)"
  shows "∏ {x. m ≤ x & x ≤ n} = ∏ {x. m ≤ x ∧ x ≤ n - 1} * n"
 proof-
  let ?A = "{m .. n}"
  let ?B = "{m .. n - 1}"
  let ?C = "{n}"
  have abc: "finite ?B" "finite ?C" "?B Int ?C = {}" "?B Un ?C = ?A"
  using n by auto
  from setprod_Un_disjoint[OF abc(1-3), of "%x. x"]
  show ?thesis unfolding abc(4) setinterval_iff by simp
 qed

 lemma upperBoundRangeSetEq:
    "{x. m ≤ x ∧ x < n} = {x::int. m ≤ x & x ≤ n - 1}"
  by(auto)

 lemma prodEqProdTimesLess [simp]:
  assumes n: "m - 1 < (n::int)"
  shows "∏ {x. m ≤ x & x ≤ n} = ∏ {x. m ≤ x ∧ x < n} * n"
 proof -
  show ?thesis
  by (simp only: upperBoundRangeSetEq,
      rule prodEqProdTimes, rule n)
 qed
end
```