Kopitiam – a unified IDE for developing formally verified Java programs

Hannes Mehnert
Jesper Bengtson

IT University Technical Report Series **TR-2013-167**

ISSN 1600–6100 **May 2013**

ISSN 1600–6100

ISBN 978-87-7949-304-9

Copies may be obtained by contacting:

IT University of Copenhagen
Rued Langgaards Vej 7
DK-2300 Copenhagen S
Denmark

Telephone:	**+45 72 18 50 00**
Telefax:	**+45 72 18 50 01**
Web	`www.itu.dk`

Kopitiam – a unified IDE for developing formally verified Java programs

Hannes Mehnert and Jesper Bengtson

IT University of Copenhagen
{hame, jebe}@itu.dk

Abstract. We present Kopitiam, an Eclipse plugin for certifying full functional correctness of Java programs using higher-order separation logic. Kopitiam extends the Eclipse Java IDE with an interactive environment for program verification, powered by the general-purpose proof assistant Coq. Moreover, Kopitiam includes a development environment for Coq theories, where users can define program models, and prove theorems required for the program verification.

1 Introduction

It is difficult to design software that is guaranteed to work. For most software, correctness is inferred by extensive and costly testing, but this can only prove the presence of errors, not their absence. For safety critical systems this is unsatisfactory. The ideal is to specify the desired behaviour of a program using logics and mathematics, and then formally prove that the program satisfies its specification; this guarantees that all cases have been covered and that no stone has been left unturned. Formal methods have had a renaissance in recent years, and they are being incorporated into tools that are used in different parts of the software development cycle, but they are typically highly specialised and focus on ensuring that a program satisfies a key property such as deadlock freedom, memory safety, or termination.

In this paper we present Kopitiam, an extension to the Java development environment of Eclipse that allows programmers to prove full functional correctness of Java programs. Kopitiam is designed according to the following set of design philosophies:

1. Users should be able to develop programs, their models, their specifications and their proofs of correctness incrementally and side by side.
2. The program logic must be expressive enough to reason about features such as mutable state, shared data structures and pointer aliasing.
3. The specification language must be expressive enough to describe the behaviour of all programs that we want to certify
4. All proofs that users write must be checked automatically.
5. No bug in the tool itself may break soundness – a buggy tool may fail to certify a program, but it must never claim that a defective program is correct.
6. The integrated development environment (IDE) must include the tools that software developers are accustomed to, such as compilers, profilers, unit testers, and debuggers.

Software development is an incremental process, and it is important that programs are kept synchronised with their proofs of correctness. This is ensured by point 1. The danger is otherwise that a proof proves properties about an outdated version of the program.

Point 2 requires that we support some dialect of separation logic [30]. Since its conception, separation logic has been used to great success for modular reasoning about programs written in languages using shared data, pointers, aliasing, and destructive updates, including object-oriented languages [8,26,27,34]. By *modular reasoning* we mean that the specification of a program can constrain itself to the state that the program actually acts upon, and not the entire proof state.

Point 3 requires that the proofs are developed interactively as the heuristics required to find them will generally be undecidable. In practice, this requires that the user has access to an interactive proof assistant such as Coq [32], Isabelle [25], or HOL [15]. These theorem provers support a higher-order logic that is expressive enough to reason not only about specification languages but also about the semantics of programming languages. Combined, this allows them to certify theorems that prove that programs satisfy their specifications with respect to the semantics of the programming language. Another distinct advantage of the interactive proof assistants is that they have a kernel that automatically certifies the proofs produced by the user, as required by point 4. Moreover, this kernel

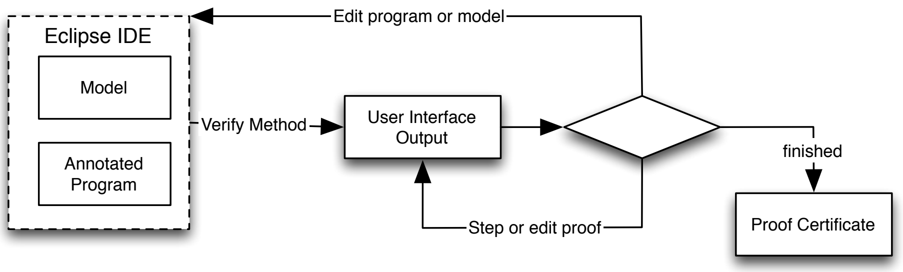


Fig. 1. Kopitiam workflow. The user writes an annotated program in the Java perspective, and a model of the program in the Coq perspective. Each method is verified one at a time. The user steps through the statements of the method, using inline Coq commands to update the proof state where necessary. Kopitiam automatically produces a certificate of correctness when all methods have been verified.

is very small and is the most complicated piece of code that actually has to be trusted (apart from a few minor parsing and output routines). This provides the reliability required by point 5. The kernel will reject a proof that is incorrect, even if it was produced by the theorem prover or by another part of the tool, due to a bug.

Finally, it is important that the developer has access to a state of the art development environment for the actual software development, as required by point 6. Even though our ultimate goal is to prove the absence of bugs in our programs, the process of formal verification is currently prohibitive and testing and debugging are very useful tools to get a program in such a shape that it can reasonably be assumed to be bug free before undertaking the verification effort.

Kopitiam Kopitiam is an Eclipse plugin that integrates the Java development environment of Eclipse with the general-purpose interactive proof assistant Coq. The software verification process is split into two distinct parts; proofs about the program specification, which are done in a Coq perspective, and proofs about the actual program, which are interleaved with the program itself in the Java IDE. Program verification is interactive. The user steps through the program, in the style of a debugger, proving that all prerequisites are met for every statement. The workflow is described in Figure 1.

As a back-end, Kopitiam uses Charge! [4,5], a framework for verifying Java programs using an intuitionistic higher-order separation logic in Coq. Charge! provides tactics that automatically discharge commonly occurring proof obligations, allowing the user to focus on the interesting rather than the tedious aspects of program verification.

Contributions Our main contribution is that we provide one uniform framework for formal verification of Java programs that integrates an industry-grade IDE with a general-purpose interactive proof assistant. More specifically, Kopitiam includes:

– *An extension to the Eclipse Java development environment* that allows users to annotate their code with specifications for each method (§2.2, 3.2). The user steps through the program one statement at a time, executing each symbolically and observing how the proof state changes. All steps are verified by Charge!. If necessary, these steps are interleaved with Coq code to prove that the current proof state satisfies the conditions required to make the next step.
– *A development environment for Coq theories* (§2.2, 3.1). This perspective allows users to state theorems in Coq, and construct proofs of their correctness interactively, much like Proof General, or CoqIDE. This mode is used to define the model of the program, and prove all necessary properties needed for verification.
– *Support for passing Java program variables as arguments to predicates defined in the Coq standard library* (§2.2 p. 6). This allows models and specifications to use predicates and terms that have not been designed with software verification in mind.
– *Support for proof certificates*. Once a program has been verified with Kopitiam, we produce a proof certificate (§2.2 p. 7), which contains the program and a theorem and corresponding proof that the program is correct. This proof certificate is checkable by Coq using Charge!.

```
class List {
  static class Node {
    int value;
    Node next;

    public int nodeLength() {
      int r = 1;
      Node n = next;
      if (n != null) {
        r = n.nodeLength();
        r = r + 1;
      }
      return r;
    }
  }

  Node head;
  public int length() {
    Node h = head; int r = 0;
    if (h != null) r = h.nodeLength();
    return r;
  }
  public void add (int n) {
    Node x = new Node(); x.value = n;
    Node h = head; x.next = h;
    head = x;
  }
  public void reverse () {
    Node old = null; Node lst = head;
    while (lst != null) {
      Node tmp = lst.next; lst.next = old;
      old = lst; lst = tmp;
    }
    head = old;
  }
}
```

Fig. 2. A small list library. The library has one inner Node class, which is used for each list element. The Node class allows us to differentiate between an empty list (where head is null) and the null pointer.

To the best of our knowledge, no other tool integrates an industry-grade IDE like Eclipse with an interactive proof assistant this closely. There are other tools that target software verification, but those that use interactive proof assistants either verify the programs completely in a proof assistant or generate theory files from annotated code (and thus lose the ability to keep code and proofs synchronised). Another approach is to build the entire development and verification environment from scratch, which invariably leads to fewer features than those available when building on already well-established IDEs and proof assistants. An extensive comparison with related work is presented in Section 5.

Kopitiam is released under the BSD license. Examples and installation instructions are available at `http://itu.dk/research/tomeso/kopitiam/`.

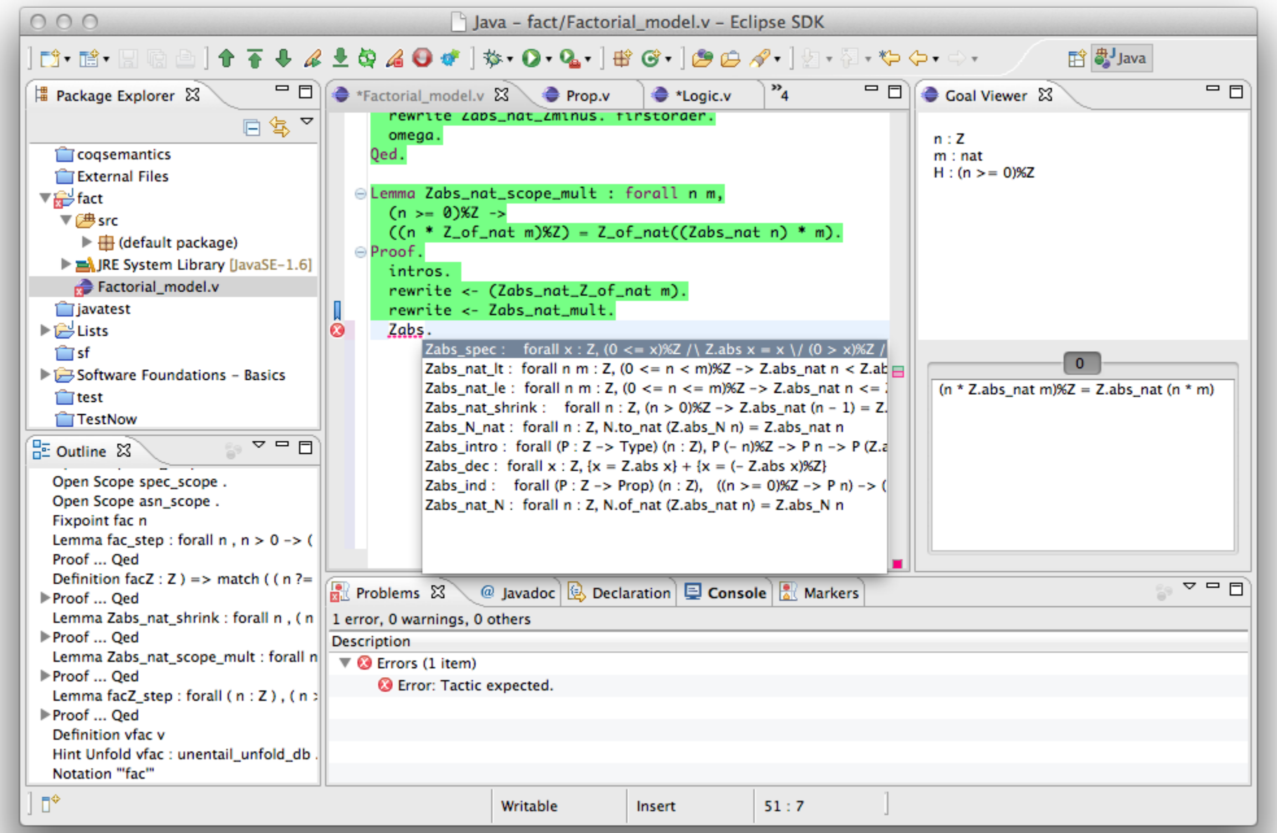
2 Using Kopitiam

We demonstrate program verification in Kopitiam by certifying a small library for linked lists with two classes: an inner class Node for the list elements and the class List for the lists themselves. The public API of the List class consists of a length method, which defers its work to the auxiliary recursive nodeLength method in the Node class, an add method that adds an element to the head of the list, and finally a method reverse for in place list reversal. The source code is presented in Figure 2.

To verify this library, we need a model for each class, and a specification for each method. The models, and the proofs about them, are developed in the Coq perspective. The specifications, and the proofs of the actual source code, are developed in the Java perspective. We discuss each in turn.

2.1 Coq perspective

The Coq perspective is a development environment for Coq theories in Eclipse. It highlights the command currently being processed, as well as the commands that have been processed by Coq, with a yellow and a green background respectively. If the proof script contains an error, the faulty code is highlighted with a red squiggly line, and the error is presented in the error window. A goal viewer shows the current proof obligation and further subgoals. Kopitiam also supports keyboard shortcuts for common tasks, such as stepping through the proofs, and syntax highlighting. We also support features not commonly found in theorem prover interfaces such as folding of proofs, an outline for Coq theories, and code completion that suggests commands and lemmas, including the ones developed by the user. The perspective is powerful enough to develop standard Coq theories, and is not limited to program verification. A screenshot is presented in Figure 3.



Java – fact/Factorial_model.v – Eclipse SDK

Package Explorer

```
coqsemantics
External Files
fact
  src
    (default package)
  JRE System Library [JavaSE-1.6]
  Factorial_model.v
javatest
Lists
sf
Software Foundations – Basics
test
TestNow
```

Outline

```
Open Scope spec_scope .
Open Scope asn_scope .
Fixpoint fac n
Lemma fac_step : forall n , n > 0 -> (
Proof ... Qed
Definition facZ : Z ) => match ( ( n ?=
Proof ... Qed
Lemma Zabs_nat_shrink : forall n , ( n
Proof ... Qed
Lemma Zabs_nat_scope_mult : forall n
Proof ... Qed
Lemma facZ_step : forall ( n : Z ) , ( n :
Proof ... Qed
Definition vfac v
Hint Unfold vfac : unentail_unfold_db .
Notation "'fac'"
```

*Factorial_model.v | Prop.v | *Logic.v

```
  rewrite Zabs_nat_Zminus. firstorder.
  omega.
Qed.

Lemma Zabs_nat_scope_mult : forall n m,
  (n >= 0)%Z ->
  ((n * Z_of_nat m)%Z) = Z_of_nat((Zabs_nat n) * m).
Proof.
  intros.
  rewrite <- (Zabs_nat_Z_of_nat m).
  rewrite <- Zabs_nat_mult.
  Zabs.
```

```
Zabs_spec :     forall x : Z, (0 <= x)%Z /\ Z.abs x = x \/ (0 > x)%Z /
Zabs_nat_lt :  forall n m : Z, (0 <= n < m)%Z -> Z.abs_nat n < Z.ab
Zabs_nat_le :  forall n m : Z, (0 <= n <= m)%Z -> Z.abs_nat n <=
Zabs_nat_shrink :    forall n : Z, (n > 0)%Z -> Z.abs_nat (n - 1) = Z.
Zabs_N_nat :  forall n : Z, N.to_nat (Z.abs_N n) = Z.abs_nat n
Zabs_intro :  forall (P : Z -> Type) (n : Z), P (- n)%Z -> P n -> P (Z.a
Zabs_dec :  forall x : Z, {x = Z.abs x} + {x = (- Z.abs x)%Z}
Zabs_ind :    forall (P : Z -> Prop) (n : Z),   ((n >= 0)%Z -> P n) -> (
Zabs_nat_N :  forall n : Z, N.of_nat (Z.abs_nat n) = Z.abs_N n
```

Goal Viewer

```
n : Z
m : nat
H : (n >= 0)%Z
```

0

```
(n * Z.abs_nat m)%Z = Z.abs_nat (n * m)
```

Problems | Javadoc | Declaration | Console | Markers

```
1 error, 0 warnings, 0 others
Description
Errors (1 item)
  Error: Tactic expected.
```

Writable | Insert | 51 : 7

Fig. 3. Screenshot of the Coq perspective in Kopitiam. To the left, the package explorer and the Coq theory outline; in the middle, the theory file and the error window; to the right, the goal viewer which has the Coq context at the top, and the current goal at the bottom. The 0 indicates that there is currently only one subgoal. In the theory file, the green background indicates how far the theory has been processed by Coq. Moreover, the term Zabs has been underlined and marked as an error since no lemma or tactic with that name exists; by pressing Ctrl-Space we obtain a list of possible completions.

Defining the program model The theoretical foundation of Kopitiam is the Charge! framework, which verifies Java-like programs in Coq using a higher-order intuitionistic separation logic. Separation logic allows users to reason about the shape and contents of the heap. For this exposition, we only require two construct specific to separation logic: the points-to predicate ($\mapsto$) and the separating conjunction ($*$). The predicate $c.f \mapsto v$ states that an object instance referenced by c has a field f that currently contains the value v; the predicate $p * q$ states that the predicate p holds for one part of the heap and the predicate q holds for another disjoint part of the heap. The $*$-operator gives rise to the characteristic frame rule of separation logic

$$\frac{\{p\}c\{q\}}{\{p * r\}c\{q * r\}}\ \textit{c does not mention r}$$

which states that a command c that runs in the state p and terminates in q, can run in an environment where a disjoint r is present as long as no command in c modifies r. In general, the Hoare-triple $\{p\}c\{q\}$ states that if the command c starts from state p and terminates, it will satisfy the state q.

The semantics of Java in Charge! is untyped. A single type val in Coq represents Java types. Separation logic predicates have the type asn. There is also a typeof predicate, where typeof p C states that the dynamic type of the object reference p is the class C.

For our list library, we define one model for the inner Node class, and one for the List class. Throughout the paper, we use standard mathematical notation in Coq code, and not the ASCII code used by Charge!.

```
<% lvars: $xs$ : list val %>
<% requires: `List_rep "this"/V `$xs$ %>
<% ensures: `List_rep "this"/V (`cons "n"/V `$xs$) %>
public void add (int n) { . . . }

<% lvars: $xs$ : list val %>
<% requires: `List_rep "this"/V `$xs$ %>
<% ensures: `List_rep "this"/V `(rev $xs$) %>
public void reverse () { . . . }

<% lvars: $v$ : val, $xs$ : list val %>
<% requires: "this"/V.`next ↦ `$v$ ∗ `Node_rep `$v$ `$xs$
<% ensures: "r", "this"/V.`next ↦ `$v$ ∗ `Node_rep `$v$ `$xs$ ∧
                    "r"/V == `((length $xs$) + 1)%>
public int nodeLength () { . . . }

<% lvars: $xs$ : list val %>
<% requires: `List_rep "this"/V `$xs$ %>
<% ensures: "r", `List_rep "this"/V `$xs$ ∧ "r"/V == `(length $xs$) %>
public int length () { . . . }
```

Fig. 4. Specification of the methods in the list library. Note how they all use the backtick operator (`), or the /V notation, so that predicates that normally take values as arguments take program variables instead. This applies not only to our own List_rep and Node_rep predicates, but also to functions from the Coq standard library like cons. The ensures clauses in nodeLength and length both have the return variable "r", which indicates that any value returned will be assigned to this variable.

```
Fixpoint Node_list (p : val) (lst : list val) : asn :=
  match lst with
    | nil ⇒ p = null
    | x :: xs ⇒ typeof p Node ∧ ∃v : val. p.value↦x ∗ p.next↦v ∗ Node_list v xs
  end.
```

Lists are modelled using the lists from the Coq standard library. The predicate Node_list p lst is defined recursively over lst and creates one instance of Node on the heap for every element in lst, where every instance points to the next respective element in the list.

The predicate List_rep p lst states that p has the dynamic type List and that the head field of p points to an object modelled by the Node_list predicate.

```
Definition List_rep (p : val) (lst : list val) : asn :=
    typeof p List ∧ ∃h : val. p.head↦h ∗ Node_list h lst
```

Proof development The Coq perspective is also used to develop the meta-theoretical properties that we require for the program model. For this list library, we need a lemma that states that the only list that can model the null-pointer is the empty list.

Lemma lst_ptr_null: $\forall k\ \mathit{lst}.\ k = \mathsf{null} \longrightarrow (\mathsf{Node\_list}\ k\ \mathit{lst} \vdash \mathit{lst} = [])$
Proof.
    . . .
Qed.

This lemma states that if k is null and k is modelled by a list *lst*, then *lst* must be the empty list. This lemma is proven in Coq by case analysis on *lst*.

2.2 Java perspective

The Java perspective of Kopitiam is an extension of the Java editor. We have extended the syntax with antiquotation brackets <% and %>, between which Kopitiam-specific code is placed. These brackets and their contents are ignored by the rest of Eclipse and all of its standard functionality is maintained. We use the Java perspective to write programs and their specifications, and ultimately prove that the programs satisfy their specifications.

Specifications Specifications of methods have the form

<% lvars: $x_1 : T_1, \ldots,\ x_n : T_n$ %>
<% requires: $P\ x_1\ \ldots\ x_n$ %>
<% ensures: $Q\ x_1\ \ldots\ x_n$ %>

where x_1 to x_n are logical variables universally quantified over P and Q, T_1 to T_n are their types, P is the precondition and Q is the postcondition of the method. Here T_1 to T_n are standard Coq types, and P and Q are Coq predicates in our assertion logic; typically the model of each class will contain the building blocks required to construct these predicates.

Specifications can also mention the program variables that each method takes as arguments, and the this pointer, but making this compatible with the predicates we defined for the program model is not entirely straightforward. Program variables are strings, and their values are stored on the stack. The List_rep predicate has the type val $\rightarrow$ list val $\rightarrow$ asn. We would like to be able to declare an assertion List_rep "p" xs stating that the program variable p points to a list xs, but this will not type check in Coq as "p" has the type string and not val. In Charge! this problem is solved using an environment monad [4] that allows assertions to be parametrised by a stack, and all program variables to be evaluated before being used in a predicate. We will not go into the exact details here, but by prefixing any Coq predicate and its arguments with a backtick (`), we allow these predicates to use the stack to look up the value of any program variable. We also have the expression "p"/V that returns a function that, given a stack, looks up the program variable "p" on the stack. The List_rep predicate is then written as `List_rep ("p"/V) (`$xs$), where the parentheses can be omitted. Note that `xs is evaluated to xs since xs contains no program variables, but the backtick is still required to make the term type check. One important observation is that we can apply the environment monad to any Coq type, which allows us to use the standard built-in libraries of Coq with our specifications, even though they were not originally designed with program variables in mind.

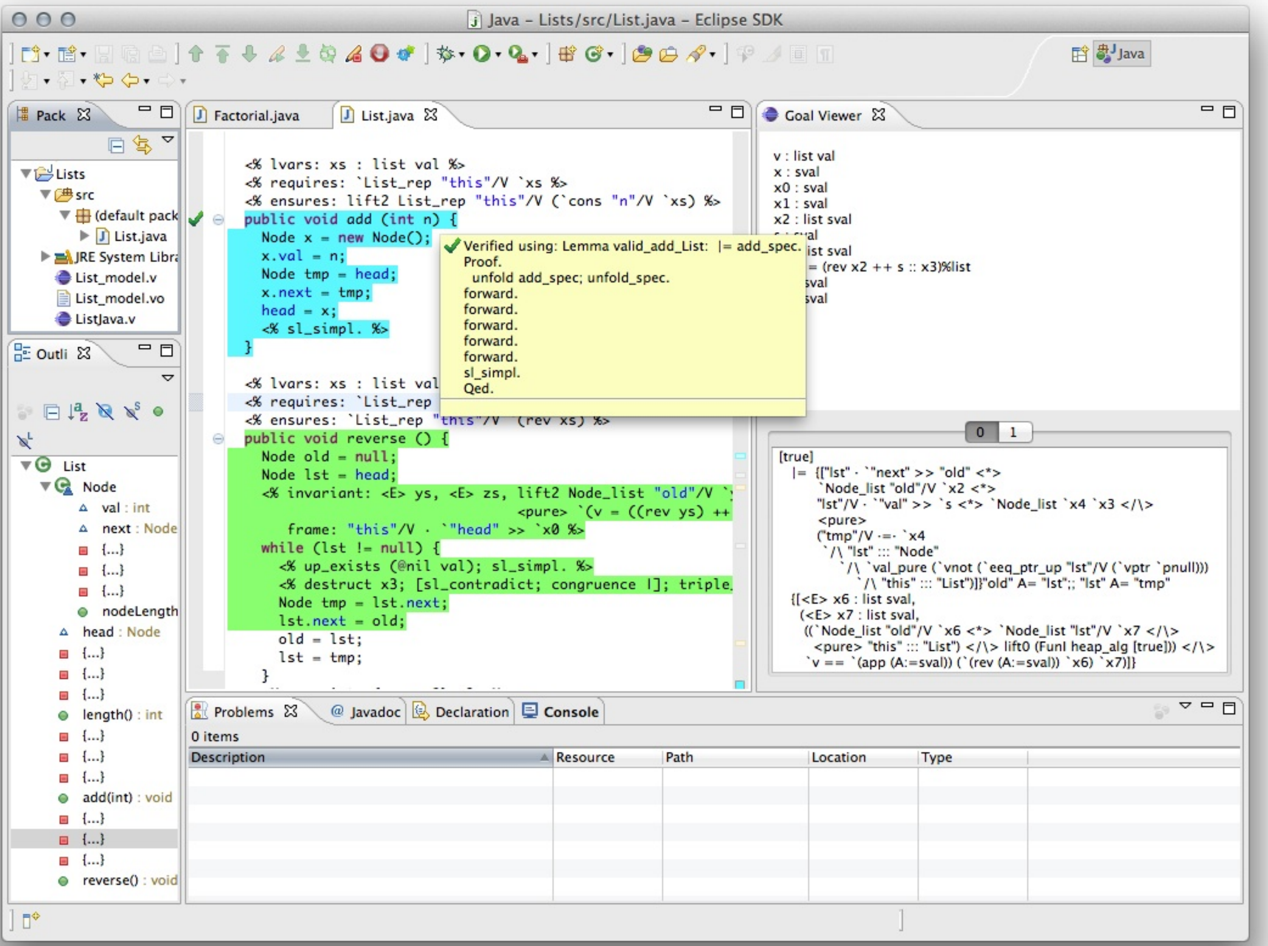
The specifications for all methods in our list library are presented in Figure 4. The specification for nodeLength uses the Node_rep predicate directly since this method is defined in the Node class. The other methods are defined in the List class, and use the List_rep predicate for their specifications. A user of this library would only need to know about the List_rep predicate.

Formal verification Classes are verified method by method. By right-clicking on a method and selecting Verify Method from the menu, we enter a mode that lets us prove formally that the method satisfies its specification. A screenshot is presented in Figure 5. Charge! includes a semantics for Java that comes with an inference rule for every statement type. Methods are proved correct by stepping through them one statement at a time, using a tactic named forward, which is inserted automatically for each statement. The forward tactic checks whether the precondition is satisfied, and if so executes the next command symbolically and updates the program state. If the precondition is not satisfied, the user can insert standard Coq code in antiquotes between statements to change the state in such a way that the next step is enabled.

The Java perspective uses the same goal viewer as the Coq perspective, and displays a Hoare triple $\{P\}c\{Q\}$ where c is the remaining statements, and the precondition P is the current program state. The predicate Q is most often the postcondition of the method, but it is also used at the end of loops and conditional branches.

Loops are annotated with two predicates. The first is an invariant, which is a predicate that must hold on loop entry and on all iterations of the loop. The second is a frame, which is a predicate that is part of the current proof state and which is not required to prove the loop body, but which is required to prove the remaining code of the method. The frame is only available outside of the loop. This is achieved by using to the frame rule from page 4.

The proofs of the different methods in the list library vary in complexity. The simplest one is the add method, which requires only its specification and a single sl_simpl annotation. The proofs for the nodeLength and length methods are relatively straightforward. At each method call, the precondition of nodeLength must be established, but the fact that the method is recursive makes no difference to the complexity of the proof. The most complex proof is the one for the reverse method, which is presented in Figure 6. This proof nicely demonstrates the way software is verified using Kopitiam. Manual proofs are required in key places, but these can often be kept small and concise. In particular, any properties pertaining to the model should be proved as separate lemmas in the model file.



```
<% lvars: xs : list val %>
<% requires: `List_rep "this"/V `xs %>
<% ensures: lift2 List_rep "this"/V (`cons "n"/V `xs) %>
public void add (int n) {
  Node x = new Node();
  x.val = n;
  Node tmp = head;
  x.next = tmp;
  head = x;
  <% sl_simpl. %>
}

<% lvars: xs : list val
<% requires: `List_rep
<% ensures: `List_rep "this"/V `(rev xs) %>
public void reverse () {
  Node old = null;
  Node lst = head;
  <% invariant: <E> ys, <E> zs, lift2 Node_list "old"/V `
                        <pure> `(v = ((rev ys) ++
     frame: "this"/V · `"head" >> `x0 %>
  while (lst != null) {
    <% up_exists (@nil val); sl_simpl. %>
    <% destruct x3; [sl_contradict; congruence |]; triple
    Node tmp = lst.next;
    lst.next = old;
    old = lst;
    lst = tmp;
  }
```

```
Verified using: Lemma valid_add_List:  |= add_spec.
Proof.
  unfold add_spec; unfold_spec.
forward.
forward.
forward.
forward.
forward.
sl_simpl.
Qed.
```

```
v : list val
x : sval
x0 : sval
x1 : sval
x2 : list sval
     ist sval
   = (rev x2 ++ s :: x3)%list
     sval
     sval
```

```
[true]
  |= {["lst" · `"next" >> "old" <*>
      `Node_list "old"/V `x2 <*>
      "lst"/V · `"val" >> `s <*> `Node_list `x4 `x3 </\>
      <pure>
      ("tmp"/V ·=· `x4
       `/\ "lst" ::: "Node"
          `/\ `val_pure (`vnot (`eeq_ptr_up "lst"/V (`vptr `pnull)))
            `/\ "this" ::: "List")]}"old" A= "lst";; "lst" A= "tmp"
 {[<E> x6 : list sval,
   (<E> x7 : list sval,
    ((`Node_list "old"/V `x6 <*> `Node_list "lst"/V `x7 </\>
      <pure> "this" ::: "List") </\> lift0 (FunI heap_alg [true])) </\>
   `v == `(app (A:=sval)) (`(rev (A:=sval)) `x6) `x7)]}
```

Fig. 5. Screenshot of the Java perspective. In the middle, the Java editor with antiquotes for specifications and proofs. The green background indicates commands processed by Coq. The blue background indicates a verified method; its tooltip contains the lemma statement and its proof, including the calls to the forward tactic. To the right, the goal viewer, displaying the Hoare triple of the remaining loop body.

Proof Certificate For each method a validity theorem is produced, which states that the implementation fulfills its specification. Once this theorem is discharged, the method in question is marked with a green checkmark and light blue background, as shown Figure 5.

Once all methods of a class have been verified, Kopitiam generates a proof certificate. This certificate contains the Java program code as Coq definitions for all methods and classes. The theorem that the entire program is correct is derived automatically from the theorems that state that each individual method is correct. Ultimately, we obtain one theorem of program correctness which can be checked completely by the Coq kernel using Charge! independently of Kopitiam.

3 Implementation

Kopitiam is not a stand-alone application, but an Eclipse plugin. Eclipse is written in Java, and thus runs on the JVM. Kopitiam is developed in Scala, a functional object-centered language which also runs on the JVM. In this section, we discuss two important aspects of our development. The first is how Kopitiam communicates with Coq and stays responsive while Coq is working on a proof. The second is how we extend Java's syntax with antiquotes

```
void reverse () {
  Node old = null; Node lst = this.head;
  <% invariant: ∃ys zs. `Node_list `"old" `ys ∗ `Node_list `"lst" `zs ∧
                      `(v = ((rev ys) ++ zs))
      frame: "this"/V.`head ↦ `x0 ∧ `typeof "this"/V `List%>
  while (lst != null) {
    //Proof of the loop invariant for loop entry follows
    <% up_exists (@nil val); sl_simpl. %>
    <% destruct x3; [sl_contradict; congruence |]; triple_nf. %>
    Node tmp = lst.next; lst.next = old; old = lst; lst = tmp;
  }
  <% ... Proof of the loop invariant for the loop body (3 lines) ...%>
  this.head = old;
  <% ... Proof of the postcondition (3 lines) ...%>
}
```

Fig. 6. Proof of the reverse method. User guided proofs are required to prove the loop invariant for loop entry and for the loop body, and to prove the postcondition. Coq automatically introduces new logical variables for the binders we use. The logical variables v in the invariant represents xs from the specification, and $x0$ in the frame represents h from the definition of List_rep (page 5).

while maintaining support of the Eclipse tools that work on regular Java projects. We will also briefly discuss parsing of Coq code, how we translate Java into Coq definitions, what measures we take to increase performance, and finally how the Verify Method action is implemented.

3.1 Coq Interaction

Coq does not provide a standard API. All communication is handled via the `coqtop` binary, which in turn communicates solely via the standard input, output and error streams. Thus all data must be sent and received as strings.

The main obstacle to overcome when interfacing with Coq is to achieve a responsive and reliable communication. Coq often takes a long time to process commands and it is important that Kopitiam does not freeze while Coq is working. Moreover Coq only responds to complete commands, and care has to be taken to avoid a state where Coq waits for more input, while Kopitiam waits for a response from Coq.

We solve these problems by designing an asynchronous publish/subscribe system using Scala actors, shown in Figure 7. This allows users to send commands without the IDE being blocked while waiting for a response. Kopitiam has the central message distributor PrintActor that maintains a set of subscribers. The PrintActor receives a message as a string from Coq, parses it to a more structured form, and redistributes this parsed message to all subscribers. Kopitiam encapsulates communication with Coq within the CoqTop singleton object. This object provides a single method `writeToCoq`, which takes a string and transforms it to a complete command (by adding the terminating '.' if necessary) before sending the command to Coq.

The subscribers of the message distributor are shown on the right of Figure 7, we describe them in ascending order. The DocumentState object tracks Coq's state. Coq maintains a unique number for each successful state update, and DocumentState stores a mapping from a position in the source file to Coq's state. This makes it possible to step back an arbitrary number of commands to a previous state. The GoalViewer updates the goal viewer in the Coq and Java perspectives; the CoqEditor colors the background of the processed Coq commands and manages error reporting. The ContentAssistant stores names of proven lemmas in the current file and uses them to suggest completions to the user. The ContentAssistant also uses Coq's `SearchAbout` command to find possible completions from other files. Finally, users interact with CoqTop every time they step through a proof or program.

This design for interaction with Coq is extensible. The Java perspective in Kopitiam has a lot in common with the Coq perspective; the user performs actions such as stepping through code, writing proof scripts, and retracting proof state. Nearly all code for communicating with Coq has been reused between the two perspectives. The only perspective-specific code handles the positioning information in the DocumentState object, since the syntax of Coq files and Java files differ.

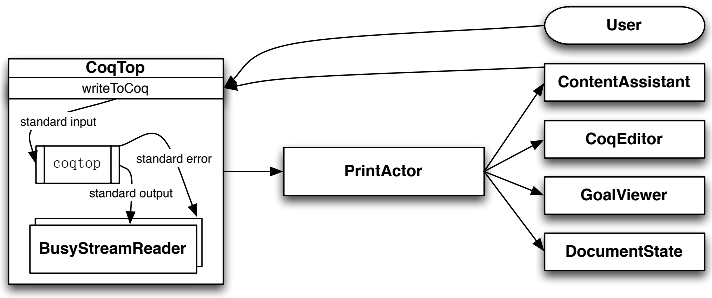


Fig. 7. The singleton object CoqTop encapsulates the communication with `coqtop`. The method `writeToCoq` sends a message to the standard input stream of `coqtop`. The CoqTop singleton starts a process and connects the standard output and error streams to a BusyStreamReader instance each. When a BusyStreamReader instance receives a message, it forwards that to the singleton object PrintActor. This is the central distributor of messages, and first parses the received string into a more structured object and sends that asynchronously to all subscribers on the right side.

3.2 Antiquotes

We use antiquotes to extend the Java syntax to handle method specifications, which are written before method definitions, and proof scripts, which are written inside method bodies and interleaved with Java statements.

In order to integrate these antiquotes with the Java syntax we extend the Java lexer and parser in Eclipse. Since neither the lexer, nor the parser, provide an off-the-shelf extension mechanism, we use aspect-oriented programming to modify the bytecode of Eclipse at load-time. Our extension creates a dummy node for each antiquote in the abstract syntax tree of the Java programs. The developer can still use all standard Eclipse tools like debuggers, compilers, profilers, and unit testers, since the dummy nodes are ignored by Eclipse. The extensions to the Eclipse parser and lexer are available as a separate Eclipse plugin[1].

We evaluated three other possible approaches to embed specifications and proof script inside of programs: annotations, comments, and static method calls.

Java annotations are only allowed before methods, not inside method bodies, and would only support method specifications. Type annotations are a recent development that allow annotations at type occurrences, but since they also can not be placed inside the method bodies they do not solve the problem.

Java comments are allowed anywhere, making them a good candidate. However, the Eclipse Java lexer throws comments away. If we used comments, we would have to modify the internal behaviour of the Java lexer (to keep the embedded statements) and parser (to make sense of these embedded statements). This would make our implementation prone to breakage with respect to future releases of Eclipse.

In an earlier version of Kopitiam [21] we represented method specifications and proofs in the Java source code as strings in calls to dummy static methods on a dummy Coq class. This was cumbersome and all commands had to be passed as strings. Moreover, method calls are not allowed outside method bodies, where specifications are written.

3.3 Parsing of Coq code

Kopitiam needs to parse Coq code to support features like syntax highlighting, an outline, and proof folding. The folding hides proof script of a lemma between the Proof and Qed keywords, similarly to how method bodies are folded in Eclipse.

Rather than having a complete parser for Coq's grammar, we first split Coq code into sentences of complete commands. Each sentence is then parsed individually to check for the keywords that are required by the highlighter, the outline, and the folder. Parsing the entire Coq code is not realistic as Coq allows users to dynamically change its grammar on the fly using a powerful notation mechanism. Since even compiled Coq binaries can include such changes, creating an external parser for complete Coq commands is impractical. Our solution allows us to only check each command for the keywords that we require to build the functionality we need. We leave the complete parsing of each individual command required for proof checking to Coq.

[1] https://github.com/hannesm/KopitiamAspects

3.4 Java translation

Kopitiam translates Java into the subset of Java formalized in Charge!. This support includes read and write operations for fields and variables, static and dynamic method calls (including recursion), while loops, conditionals, and object allocation. The latter creates a new instance of an object and allocates memory for its fields on the heap.

The translation is implemented using the visitor pattern on the Eclipse abstract syntax tree. During the traversal, Kopitiam issues warnings about missing specifications and Java code which it cannot translate to our core subset.

3.5 Verify Method

The Verify Method action in the Kopitiam user interface requires that the theory file containing the program model has been compiled, and that Coq definitions have been created for all method specifications. After these things are done, a validity theorem for the method is generated and the user is prompted to interactively verify the correctness of the method.

To increase responsiveness Kopitiam is optimised to load theories, including the program model, only when needed and unloading them only when they are modified. Moreover, Kopitiam adopts a local translation technique that translates modified code and splices it into the already translated program. The whole code is retranslated only when major revisions are made.

4 Discussion and future work

The most challenging aspects of developing Kopitiam are done. The difficult part has been figuring out how to integrate the Java IDE of Eclipse with Coq, but that aspect is complete and working well. We improve significantly on our previous release [21] where Kopitiam only supported translation of Java code into Coq definitions which had to be verified completely in the Coq perspective. For this release, the Java perspective is entirely new and we are able to conduct proofs directly within the Java editor. The Coq perspective has also gained several key features such as proof completion, syntax highlighting and the proof outline.

There are things we wish to improve, both on the IDE side (for usability reasons) and in the back-end Charge! (for performance and expressiveness reasons). Currently, programs must fit into one file. The subset of Java we support is defined in Section 3.4. Additionally, these further restrictions are imposed on the language: only a single return statement at the end of the method body is supported; the primitive types must be integers, booleans, and object references; finally, class-to-class inheritance and interfaces are not supported.

Adding more language features is a two-step process. Charge! has to be developed to include the theoretical results required, and Kopitiam must be extended to allow easy access to these results. We have developed theories to reason about inheritance using interfaces [5], and we plan to add support for this in the near future. We are also making Charge! modular by adding support for unimplemented specifications and program composition. This will allow classes to be verified independently and then combined to form a complete program.

There are three aspects of the user interface that require attention. The first is the way we write specifications, frames, and loop invariants. Currently, they are passed as raw data to Charge!. We plan to support a domain specific language in the antiquotes, with proper syntax highlighting, that hides implementation details from the users, such as the environment monad, as seen in Figure 6, namely fresh names, backquotes and /V. Secondly, Kopitiam generates fresh names when quantifiers are moved to the Coq context. These names are typically not the ones chosen by the user, but ones generated by Charge!, which can also be seen in Figure 6. Charge! uses higher-order abstract syntax to handle binders, which makes it impossible to get a handle on their names in Coq. Two ways around this are either to change how binders are modelled in Charge! or to extend the functionality of Coq to allow these bound names to be retrieved. Finally, we plan to redesign the goal viewer in the Java perspective. Currently, both the Coq and the Java perspectives use the same goal viewer, and the user sees the goal as a complete Hoare triple, which is the internal representation of the program in Charge! (see Figure 5). The ideal is to only see the precondition of the triple, which is the current program state, properly split into views describing the heap, and views describing the stack. This will make working in Kopitiam much like using a debugger, where users step through the program and are presented with a clear view of the program state.

We still have a few performance issues. These are mostly related to the Coq back-end. Charge! handles the symbolic execution of commands as the user steps through the program in the Java editor, and this can be slow if the next command has a precondition that is difficult to prove. We plan to rewrite our heuristics using reflective tactics, similar to the ones used by Chlipala in his Bedrock framework [12], and this will speed up proof search considerably.

The largest case study we have verified using Charge! is the snapshotable trees library from the C5 library [22]. Currently, this formalisation only exists as a Coq theory and a next logical step is to certify the library using Kopitiam.

We are currently using Kopitiam in a master course on software verification based on the *Software Foundations* book [29][2] by Pierce et al. So far, student feedback has been very positive and the Coq perspective of Kopitiam is proving capable of dealing with all material and exercises presented in the book.

5 Related work

Several tools for program verification of varying complexity have appeared in the last decades. Like in Kopitiam, source code is typically annotated with specifications, after which the tools check if these specifications are satisfied.

Many tools strive for automation, which by necessity limits their expressivity. Typically, they produce a large number of lemmas that are automatically discharged by an automatic theorem prover, such as Z3 [24] or SPASS [35]. Examples of such tools are Spec♯ for C♯ [2] and ESC/Java2 for Java [13]. A few tools also use separation logic, like Smallfoot [6], Space Invader [11], SLAyer [7], and jStar [14].

To verify full functional correctness of programs, interactive techniques are required. The options that have been explored have followed one of three paths. The first is to generate proof obligations from annotated programs, feed them to an interactive proof assistant and have the user discharge the proof obligations manually. The second is to write the program, its specifications, and the proof of correctness entirely in an interactive proof assistant. Finally, a development environment for verified software can be developed from the ground up, possibly hooking into external proof assistants as required.

The first approach is used by tools like Jahob [19] and Loop [33]. Loop translates JML-annotated Java programs to the proof assistant PVS, which is then used interactively to discharge the goals. Jahob uses its own specification language and uses both interactive proof assistants (Isabelle and Coq) and automatic theorem provers (e.g. SPASS, CVC3, and Z3) to discharge the goals. The disadvantage to this approach is that it is difficult to keep the proof script synchronised with the program – once the script has been generated, any changes to the program will not automatically propagate to its proof and there is a danger that proofs refer to old instances of the programs. Also, neither tool uses separation logic, making it difficult to reason modularly about programs with mutable state.

The second approach (programming directly in a proof assistant) has had great success. Two notable examples are the verified L4 micro-kernel by Klein et al. [18] and the verified optimising C compiler CompCert by Leroy [20]. These two projects have set the standard for state-of-the-art formally verified software today, but each also required a Herculean effort to complete. Both projects verify around 7000 lines of code, yet their proof sizes are orders of magnitude larger (around 50000 lines of code for CompCert and 200000 for the L4 kernel); the L4 kernel required around 25 man years to complete, CompCert around 10. For the L4 kernel, the Simpl library by Schirmer [31] was used to translate C code to Isabelle. The CompCert compiler is written in the functional language Gallina, which is used inside Coq, and then extracted to an executable OCaml program.

The third approach (building the development environment from scratch) has been used for designing development environments such as KeY [1], Verifast [16] and Why3 [10]. KeY and Verifast have both been used successfully to verify Java and JavaCard programs [23,28]. They both use first-order logics (Dynamic logic for KeY and separation logic for Verifast), and they both use custom-made IDEs and interactive verification environments. Verifast integrates Z3 to discharge many proof obligations automatically. Why3 certifies programs written in the functional program WhyML, which can be extracted to executable OCaml code. It is the only one of these three tools that hooks into a general-purpose interactive proof assistant (Coq) to allow users to discharge proof obligations interactively. Why3 also hooks into a wide range of automatic theorem provers to help discharge proof obligations automatically. The advantage of this approach is that a lot of work is done automatically by the tool. The disadvantage is that these external provers are treated as trusted oracles which dramatically increases the size of the code base we have to trust as we have to assume that these tools are bug-free, as opposed to trusting only the Coq kernel. The Sledgehammer tool for Isabelle by Blanchette et al. [9] is designed to hook automatic theorem provers into interactive ones in a safe manner by providing techniques for the interactive proof assistant to replay the proofs generated by the automatic ones, but these techniques have not yet been implemented in Coq.

The work that mostly resembles ours comes from the Mobius project and Mobius PVE [3], which integrates several verification environments for Java with Eclipse. It includes the JACK framework that takes JML-annotated

[2] `http://www.cis.upenn.edu/~bcpierce/sf/`

Java code, generates proof obligations using a weakest precondition calculus, and discharges these proof obligations either with automatic or interactive proof assistants, including Coq. Mobius PVE comes with the ProverEditor (previously called CoqEditor) perspective for Eclipse that, like Kopitiam, allows users to develop Coq theories [17]. By using this perspective, JACK keeps the whole development cycle in Eclipse.

Comparison Kopitiam sets itself apart from all other tools in that it is based on separation logic, it is hooked up to a general-purpose interactive proof-assistant, and it allows programs to be written and verified in an industry-grade IDE. All these points are important. Tools not based on separation logic are unlikely to scale to larger programs; moreover, extending on general-purpose well maintained frameworks like Coq and Eclipse, both of which are arguably leaders in their respective fields, is preferable to building custom-made versions of our own; designing any of these two systems from scratch would be full fledged research and engineering projects in their own right.

Of the interactive tools listed above, only Verifast is based on separation logic. Jahob, JACK and Loop all generate proof obligations for general-purpose proof assistants but these are all stored in separate files and not actively kept synchronised with the code. Only JACK uses Eclipse, whereas Verifast, KeY and Why3 use their own custom-built IDEs.

6 Conclusions

Kopitiam provides the closest integration to date between an industry-grade IDE and a general-purpose proof assistant. The ability to step through the source code of a program when developing its proof is instructive, and provides an intuitive interface for programmers and proof developers alike. Moreover, Kopitiam provides a development environment for Coq theories where users can define program models, postulate lemmas and theorems, and prove these directly in Eclipse. Kopitiam also generates certificates, checkable by Coq, for each verified program. Finally, since we use an expressive higher-order separation logic we are able to model both the semantics of the Java programming language as well as the Java programs themselves in Coq. This means that the proof certificates are unified theorems, checkable by the Coq kernel, that guarantee that a verified program follows its specifications with respect to the operational semantics of our model of Java. This ensures that the trusted code base is kept very small. The only two things we have to trust are the model of Java in Charge! and the Coq kernel.

Acknowledgement Thanks to Peter Sestoft, Joseph Kiniry, Lars Birkedal, Fabrizio Montesi, Jasmin Blanchette, and Filip Sieczkowski for their valuable feedback on this paper. We want to thank David Raymond Christiansen, Alec Faithfull, Daniel Dam Freiling and Mads Hartmann Jensen for contributions to the Kopitiam source code. Thanks to Stephan Herrmann for helping out with the Eclipse parser and lexer internals.

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*, 4:32–54, 2005.
2. M. Barnett, R. DeLine, M. Fähndrich, B. Jacobs, K. R. M. Leino, W. Schulte, and H. Venter. The Spec# programming system: Challenges and directions. In *VSTTE*, pages 144–152, 2005.
3. G. Barthe, L. Beringer, P. Crégut, B. Grégoire, M. Hofmann, P. Müller, E. Poll, G. Puebla, I. Stark, and E. Vétillard. MOBIUS: Mobility, ubiquity, security. In *TGC*, pages 10–29, 2006.
4. J. Bengtson, J. B. Jensen, and L. Birkedal. Charge! – a framework for higher-order separation logic in Coq. In *ITP*, pages 315–331, 2012.
5. J. Bengtson, J. B. Jensen, F. Sieczkowski, and L. Birkedal. Verifying object-oriented programs with higher-order separation logic in Coq. In *ITP*, pages 22–38, 2011.
6. J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2006.
7. J. Berdine, B. Cook, and S. Ishtiaq. SLAyer: Memory safety for systems-level code. In *CAV*, pages 178–183, 2011.
8. B. Biering, L. Birkedal, and N. Torp-Smith. BI Hyperdoctrines and Higher-Order Separation Logic. In S. Sagiv, editor, *ESOP*, volume 3444 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2005.
9. J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending Sledgehammer with SMT solvers. In *CADE*, pages 116–130, 2011.
10. F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, Wrocław, Poland, August 2011.

11. C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Space invading systems code. In *LOPSTR*, pages 1–3, 2008.
12. A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI*, pages 234–245, 2011.
13. D. R. Cok and J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML – progress and issues in building and using ESC/Java2. In *Construction and Analysis of Safe, Secure and Interoperable Smart Devices: International Workshop*. Springer, 2004.
14. D. Distefano and M. J. Parkinson. jStar: towards practical verification for Java. In G. E. Harris, editor, *OOPSLA*, pages 213–226. ACM, 2008.
15. M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, New York, NY, USA, 1993.
16. B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods*, pages 41–55, 2011.
17. J. R. K. Julien Charles. A lightweight theorem prover interface for Eclipse. In *UITP*, 2008.
18. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In J. N. Matthews and T. E. Anderson, editors, *SOSP*, pages 207–220. ACM, 2009.
19. V. Kuncak and M. C. Rinard. An overview of the Jahob analysis system: project goals and current status. In *IPDPS*. IEEE, 2006.
20. X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
21. H. Mehnert. Kopitiam: Modular incremental interactive full functional static verification of Java code. In *NASA Formal Methods*, pages 518–524, 2011.
22. H. Mehnert, F. Sieczkowski, L. Birkedal, and P. Sestoft. Formalized verification of snapshotable trees: Separation and sharing. In *VSTTE*, pages 179–195, 2012.
23. W. Mostowski. Fully verified Java card API reference implementation. In *VERIFY*, 2007.
24. L. D. Moura and N. Bjørner. Z3: An efficient SMT solver. In *In Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.
25. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: a Proof Assistant for Higher-Order Logic*, volume 2283. 2002.
26. M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In J. Palsberg and M. Abadi, editors, *POPL*, pages 247–258. ACM, 2005.
27. M. J. Parkinson and G. M. Bierman. Separation logic, abstraction and inheritance. In G. C. Necula and P. Wadler, editors, *POPL*, pages 75–86. ACM, 2008.
28. P. Philippaerts, F. Vogels, J. Smans, B. Jacobs, and F. Piessens. The Belgian electronic identity card: a verification case study. *ECEASST*, 46, 2011.
29. B. C. Pierce and al. *Software Foundations*. University of Pennsylvania, July 2012.
30. J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.
31. N. Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In *LPAR*, pages 398–414, 2005.
32. The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V7.3*, May 2002.
33. J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In *TACAS*, pages 299–312, 2001.
34. C. Varming and L. Birkedal. Higher-Order Separation Logic in Isabelle/HOLCF. *Electronic Notes in Theoretical Computer Science*, 218:371–389, 2008.
35. C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. SPASS version 3.5. In *CADE*, pages 140–145, 2009.