jStar-eclipse: an IDE for Automated Verification of Java Programs

Daiva Naudžiūnienė
University of Cambridge
da319@cam.ac.uk

Matko Botinčan
University of Cambridge
mb741@cam.ac.uk

Dino Distefano
Queen Mary Univ. of London
and Monoidics Ltd
ddino@eecs.qmul.ac.uk

Mike Dodds
University of Cambridge
md466@cam.ac.uk

Radu Grigore
Queen Mary Univ. of London
radu.grigore@eecs.qmul.ac.uk

Matthew J. Parkinson
Microsoft Research
mattpark@microsoft.com

ABSTRACT

jStar is a tool for automatically verifying Java programs. It uses separation logic to support abstract reasoning about object specifications. jStar can verify a number of challenging design patterns, including Subject/Observer, Visitor, Factory and Pooling. However, to use jStar one has to deal with a family of command-line tools that expect specifications in separate files and diagnose the errors by inspecting the text output from these tools.

In this paper we present a plug-in, called **jStar-eclipse**, allowing programmers to use jStar from within Eclipse IDE. Our plug-in allows writing method contracts in Java source files in form of Java annotations. It automatically translates Java annotations into jStar specifications and propagates errors reported by jStar back to Eclipse, pinpointing the errors to the locations in source files. This way the plug-in ensures an overall better user experience when working with jStar. Our end goal is to make automated verification based on separation logic accessible to a broader audience.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Formal methods

General Terms

Verification

Keywords

Automated verification, Separation logic, Java, Eclipse

1. INTRODUCTION

Software verification helps in developing more reliable programs. Although initially focused on verifying low-level mission critical programs, a variety of techniques and tools have been developed recently for verifying common everyday programs written in object-oriented programming languages [2, 4, 6]. However, bringing such techniques and tools to become a part of the standard software development process has proved difficult.

In recent years, separation logic has emerged as a highly successful approach to software verification. Separation logic enables concise specifications and the verification techniques based on it were shown to scale to large code bases such as Linux kernel [5]. However, these verification efforts have universally been led by the developers of the tools. As a result, in comparison to other approaches, e.g. Spec$^{\#}$ [1], there has been little or no effort to make separation logic tools accessible to a broader audience of software developers.

jStar [6] is a verification tool for Java programs based on principles of design-by-contract and separation logic. The aim of jStar has been to generalise and simplify the verification of Java programs. It abstracts away the details of objects implementation and can verify tricky Java programs such as commonly used design patterns, including Subject/Observer, Visitor, Factory and Pooling [7].

In this paper, we pursue the jStar agenda further by developing a developer-friendly interface to jStar. To this end, we have built a plug-in for the widely used Eclipse development environment. Our plug-in allows writing function contracts within the source code. Each time a developer saves the file, the plug-in maps the annotated Java programs to jStar specifications, and propagates jStar error reports back to the original Java program, enabling error diagnosis. This makes writing a program and verifying it in one place possible. We present **jStar-eclipse** by demonstrating two examples: a simple binary tree example and a more interesting and challenging example using Iterator and Decorator patterns.

2. JSTAR-ECLIPSE

Fig. 1 shows the high level structure of **jStar-eclipse** and its workflow. We write contracts for Java class methods as Java annotations (`@Spec` in Fig. 1). The contracts are function pre- and post-conditions written using our syntax based on separation logic. For a given Java file we extract these contracts to a specification file. Using the Soot tool [10] we convert the Java program into the Jimple intermediate representation (which is easier to analyse than Java). Finally we pass the specifications and the Jimple file to jStar which tries to verify the code against the specification.

ESEC/FSE'11, September 5–9, 2011, Szeged, Hungary.

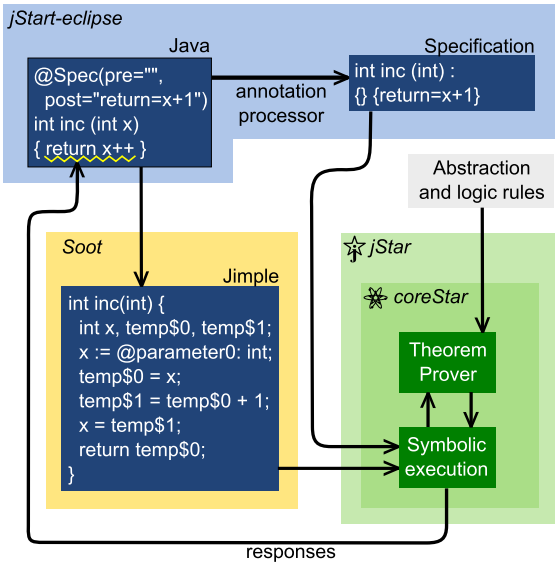

Figure 1: jStar-eclipse and its workflow

jStar is built on top of the separation logic based verification framework coreStar [3]. In addition to specifications and program code, coreStar takes collection of logic and abstraction rules (the former used by the internal theorem prover, the latter used by the internal abstract interpreter). jStar translates Jimple programs to the coreStar intermediate language. coreStar performs symbolic execution of the input program in a fix-point loop, applying abstraction and synthesising loop invariants automatically to allow termination. At the end, it checks whether each symbolic execution path satisfies the postcondition. Responses, such as error messages, are returned to the Eclipse plug-in.

2.1 New highlights of jStar-eclipse

To increase the productivity of the user, jStar-eclipse introduces several new valuable features which improve and simplify the verification process done in the jStar command line tool.

Specifications in the source file.

Java annotations are a natural way to write specifications: their purpose is to add metadata to Java source code. Writing specifications usually requires looking at the code that is being specified. Having them contained in the source code makes the verification process easier. After writing the method with its specification and saving the source file, **jStar-eclipse** automatically checks if the code satisfies its specification and shows an error message if it does not.

Error messages for verification failures.

One of the key difficulties in the verification process is understanding why the verification has failed. jStar itself returns only the part of its internal representation that it failed to verify. Relating this error back to the original Java program is difficult for non-experts. **jStar-eclipse** simplifies this debugging burden. When Java code is converted to the Jimple language, the Java source code positions are saved. This information is also kept when translating Jimple into the coreStar input language. Hence in the case of a counterexample we are able to locate the position of the Java code causing the error. The problematic source code is underlined in **jStar-eclipse**. Hovering over underlined code pops up a tooltip containing the error description.

Project-based organisation.

The files containing specifications, logic and abstraction rules are organised in the same way as Java source files. This simplifies navigation and file handling for larger projects.

3. SEPARATION LOGIC

The key benefit of separation logic is in its support for local reasoning: when we are giving a specification of a piece of code, we only need to supply information about what the code actually uses and need not to worry about the context the code is used in. To support local reasoning, separation logic introduces a new logical connective, the separating conjunction $*$, which represents disjointness of resources. The formula $P * Q$ asserts that we can split resources into two disjoint parts: one of them satisfying P and the other one Q. If we have a program C with a specification $\{P\}\ C\ \{Q\}$, then we can extend P and Q by arbitrary resource F that C does not access. This is known as the *frame rule*.

The specification $\{P\}\ C\ \{Q\}$ asserts that if we start at the state satisfying precondition P and the program C terminates then the resulting state satisfies postcondition Q. We can specify postconditions for exceptions as $\{P\}\ C\ \{Q\}\ \{exception\ class:\ E\}$. If an exception is thrown, we end up in a state satisfying postcondition E.

jStar uses *abstract predicates* to reason about object oriented programs [8, 9]. Abstract predicates can represent encapsulated properties of an object by a separation logic formula. Writing specifications in terms of abstract predicates does not expose the inner structure outside the class and thus provides us with modular reasoning and information hiding. For example, for the `Tree` class given in Fig. 2 we define an abstract predicate $Tree(x,\{sum=s\})$ which encapsulates the inner structure of the object.

4. EXAMPLES

In order to demonstrate how to verify code with **jStar-eclipse** we describe two examples. We start with a simple binary tree example and continue with a more challenging iterator example taken from the Apache Commons library.

4.1 Binary tree

Let us consider a class `Tree` representing a binary tree. Each node in a `Tree` object has left and right subtrees and a value. The method `treesum()` computes the total sum of the whole tree. Fig. 2 gives the source code, annotated with **jStar-eclipse** specifications.

We first define an abstract predicate $Tree(x,\{sum=s\})$ representing a non-empty binary tree at location x, with property sum. The definition of $Tree$ is given in the class annotation `@Predicate`. This annotation asserts that the `Tree` object has three fields: an integer field `value` that stores some value $\_v$ and two subtrees `left` and `right` that are either $null$, or store some values $\_l$ and $\_r$. We also have a derived predicate $TreeOrNull(x,\{sum=s\})$, meaning that either $x=null$ (i.e, it is an empty tree) or $Tree(x,\{sum=s\})$. To constrain $\_l$ and $\_r$ to be trees we use the $TreeOrNull$

```
@Predicate(predicate="Tree(x,{sum=s})",
  formula = "x.value |-> _v *
             x.left |-> _l * x.right |-> _r *
             TreeOrNull(_l,{sum=_ls}) *
             TreeOrNull(_r,{sum=_rs}) *
             s=((_v+_ls)+_rs)")

public class Tree {
  int value; Tree left, right;

  @Spec(pre="TreeOrNull(l,{sum=_ls}) *
             TreeOrNull(r,{sum=_rs})",
        post="Tree$(this,{sum=((v+_ls)+_rs)})")
  public Tree(int v, Tree l, Tree r) {
    super(); value = v; left = l; right = r;
  }

  @Spec(pre="Tree$(this,{sum=_s})", post="return=_s")
  int treesum() {
    int r = value;
    if (left != null) r += left.treesum();
    if (right != null) r += right.treesum();
    return r;
  }
}
```

Figure 2: Binary tree

predicate. Finally we assert that *sum* is equal to the sum of the tree, i.e., its value plus the sums of the subtrees.

Method specifications are written using the *@Spec* annotation. The specification of the class constructor asserts that, given two trees `l` and `r`, the constructor returns a tree with *sum* that is equal to the sum of the given trees and `v`. The specification of the method `treesum()` says that given a tree, it returns the value of the *sum* of the tree. By the definition of the predicate *Tree* we know that the returned value is the sum of the whole tree. These are all annotations needed to verify this example with **jStar-eclipse**.

4.2 Iterator

The second example we consider uses the Iterator and Decorator design patterns [7]. Fig. 3 gives the source code of the `FilterIterator` class[1] from Apache Commons library[2]. The `FilterIterator` decorates another `iterator` by filtering its collection values with a given `predicate`. It implements the `java.util.Iterator` interface with two methods: `hasNext()`, which returns `true` if there are more elements in the filtered collection, and `next()`, which returns the next element from the filtered collection if it exists, otherwise it throws an exception. To implement the interface, the class has two auxiliary fields: `nextObject` and `nextObjectSet`. `nextObject` works as a buffer of one cell corresponding to the next element in the collection. `nextObjectSet` says if the buffer is empty or full. We can have two possible situations as shown in Fig. 4. If `nextObjectSet` is `true`, then the first element of the filtered collection is contained in `nextObject`, otherwise, we do not use `nextObject`.

Due to limited space, we use the shorthand notation *method name: {precondition} {postcondition}* to write specifications in this example. Before specifying methods of the `FilterIterator` class, we give the specification of `java.util.Iterator` in Fig. 5. We use the abstract predicate *It(this,{collection=_c})* to abstract away the interface from the implementation. The specification of `hasNext()` says that it does not change the collection of the iterator, and the returned value is `true` if the collection has at least one element, otherwise it is `false`. The specification of `next()` says that the method returns the first element from the collection. However, it throws `NoSuchElementException` if the collection is empty.

We also need the specification of the `Predicate` interface:

```
interface Predicate {
  boolean evaluate(Object object):
  {Pred(this,{function=_f})}
  {Pred(this,{function=_f}) * return=satisfies(_f,object)}
}
```

Now we can specify the `FilterIterator` class. First we define the abstract predicate *It*[3] for this class:

```
It(x, {collection=c; under=i; underCollection=uc; pred=p;
       predFn=f; nextOb=nob; nextObSet=nobs}) =
  x.iterator |-> i * x.predicate |-> p *
  x.nextObject |-> nob * x.nextObjectSet |-> nobs *
  It(i, {collection=uc}) * i != null *
  Pred(p, {function=f}) * p != null *
  ( (nobs = true * c = cons(nob, _ct) *
    (Filter(f, uc, _ct)) * satisfies(f, nob) = true)
    ||
    (nobs = false * Filter(f, uc, c))) )
```

The definition says that we have four fields: `iterator`, `predicate`, `nextObject`, `nextObjectSet`. The field `iterator` satisfies `java.util.Iterator` interface, the field `predicate` satisfies `Predicate` interface. The last two disjuncts encode the two states in Fig. 4. *Filter(f, c1, c2)* means that if we filter the collection *c1* with the function *f* we get the collection *c2*. Specifications for `FilterIterator` methods are given in Fig. 6. The specification of `hasNext()` says that either the method returns `true` and the next object is set, or the method returns `false`, the next object is not set, and the collection is empty. The specification of `next()` says that we return the first element of the collection or we get the `NoSuchElementException` if the collection is empty. And finally, the specification of `setNextObject()` restricts calling this method only when `nextObjectSet` is `false`. The method tries to set next object and returns true if it succeeds. If it returns `false`, it must be that the underlying collection is empty.

We have used mathematical objects in the specifications, e.g. *cons*, *satisfies*, *Filter*. Here we give just their informal meanings, however to use them in the tool we have to formally define them using logic rules. Recall our informal definition of *Filter(f, c, fc)* which says that filtering the collection *c* with the function *f* we get *fc*. An example of a logic rule is an axiom saying that filtering an empty collection gives us an empty collection: $\vdash$*Filter(f, empty, empty)*.

5. CONCLUSIONS

The aim of **jStar-eclipse** was to integrate jStar within an IDE in order to support performing verification together with code writing. We have collected all that is required by the jStar command-line tool into one convenient framework. That is (1) specifications are included in the source file and automatically converted to jStar input specification files, (2) Java source is automatically converted into the intermediate representation used by jStar, and (3) error messages are

[1]For simplicity we have omitted constructors, setters, getters and remove function.

[2]http://commons.apache.org/collections/

[3]For readability we use the shorthand notation *It(x,{c,i,uc,p,f,nob,nobs})* later in the paper.

```
public class FilterIterator implements Iterator {
  private Iterator iterator;
  private Predicate predicate;
  private Object nextObject;
  private boolean nextObjectSet = false;

  public boolean hasNext() {
    if (nextObjectSet) { return true;}
    else { return setNextObject(); }
  }

  public Object next() {
    if (!nextObjectSet) {
      if (!setNextObject())
        { throw new NoSuchElementException(); }
    }
    nextObjectSet = false;
    return nextObject;
  }

  private boolean setNextObject() {
    while (iterator.hasNext()) {
      Object object = iterator.next();
      if (predicate.evaluate(object)) {
        nextObject = object;
        nextObjectSet = true;
        return true;
      }
    }
    return false;
  }
}
```

Figure 3: FilterIterator class

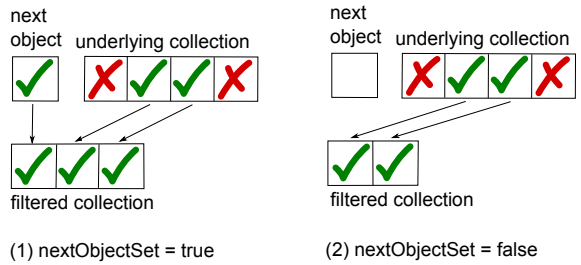

Figure 4: FilterIterator states

```
interface java.util.Iterator {
  boolean hasNext():
  {It$(this,{collection=_c})}
  {It$(this,{collection=_c}) *
   ((_c=empty * return=false) ||
    (_c=cons(_x,_ct) * return=true))}

  java.lang.Object next():
  {It$(this,{collection=_c})}
  {It$(this,{collection=_ct}) *
   _c=cons(_x,_ct) * return=_x}
  {NoSuchElementException:
   It$(this,{collection=empty}) * _c=empty}
}
```

Figure 5: Specification of java.util.Iterator

presented directly on top of the Java source code. Though **jStar-eclipse** is still at an early stage of development, already these features greatly simplify the workflow with jStar. We see development of **jStar-eclipse** as the first step

```
class FilterIterator {
  boolean hasNext():
  {It$(this,{_c;_i;_;_p;_f;_;_})}
  {It$(this,{_c;_i;_;_p;_f;_nob;_nobs}) *
   ((_c=empty * return=false * _nobs=false) ||
    (_c=cons(_nob, _ct) * _nobs=true * return=true))}

  java.lang.Object next():
  {It$(this,{_c;_i;_;_p;_f;_;_})}
  {It$(this,{_ct;_i;_;_p;_f;_;false}) *
   _c=cons(_x, _ct) * return=_x}
  {NoSuchElementException :
   It$(this,{empty;_i;empty;_p;_f;_;false}) * _c=empty}

  boolean setNextObject() static:
  {It$FilterIterator(this,{_c;_i;_;_p;_f;_;false})}
  {It$FilterIterator(this,{_c;_i;_uc;_p;_f;_;_nobs}) *
   (return=_nobs * (return=true || _uc=empty))}
}
```

Figure 6: Specification of FilterIterator

towards a user-friendly verification tool based on jStar, the first one that aims to bring separation logic based verification to a daily mainstream development.

Acknowledgments

We acknowledge funding from EPSRC H011749 (Grigore and Distefano) and H010815 (Dodds, Naudžiūnienė and Parkinson) and RAEng research fellowships (Distefano and Parkinson). Parkinson's work on jStar was carried out while at the University of Cambridge.

6. REFERENCES

[1] M. Barnett, M. Fähndrich, K. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: The spec# experience. *Communications of the ACM*, 54(6):81–91, 2011.

[2] M. Barnett, K. Leino, and W. Schulte. The Spec# programming system: An overview. *CASSIS*, 2005.

[3] M. Botinčan, D. Distefano, M. Dodds, R. Grigore, D. Naudžiūnienė, and M. J. Parkinson. coreStar: The Core of jStar. In *Boogie 2011: 1st Intl. Workshop on Intermediate Verification Languages*, 2011.

[4] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on STTT*, 7(3):212–232, 2005.

[5] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of BI-abduction. In *POPL*, 2009.

[6] D. Distefano and M. J. Parkinson. jStar: Towards practical verification for Java. In *OOPSLA*, 2008.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison–Wesley, 1995.

[8] M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In *POPL*, 2005.

[9] M. J. Parkinson and G. M. Bierman. Separation logic, abstraction and inheritance. In *POPL*, 2008.

[10] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot-a Java optimization framework. In *CASCON*, 1999.