jStar: Towards Practical Verification for Java

Conference Paper *in* ACM SIGPLAN Notices · September 2008

DOI: 10.1145/1449764.1449782 · Source: DBLP

2 authors, including:

Dino Distefano
Queen Mary, University of London
49 PUBLICATIONS **2,597** CITATIONS

SEE PROFILE

jStar: Towards Practical Verification for Java

Dino Distefano
Queen Mary, University of London, UK
ddino@dcs.qmul.ac.uk

Matthew J. Parkinson
University of Cambridge, UK
Matthew.Parkinson@cl.cam.ac.uk

Abstract

In this paper we introduce a novel methodology for verifying a large set of Java programs which builds on recent theoretical developments in program verification: it combines the idea of *abstract predicate families* [24–26] and the idea of symbolic execution and abstraction using separation logic [9]. The proposed technology has been implemented in a new automatic verification system, called jStar, which combines theorem proving and abstract interpretation techniques.

We demonstrate the effectiveness of our methodology by using jStar to verify example programs implementing four popular design patterns (*subject/observer*, *visitor*, *factory*, and *pooling*). Although these patterns are extensively used by object-oriented developers in real-world applications, so far they have been highly challenging for existing object-oriented verification techniques.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Program Verification; D.3.3 [*Programming Languages*]: Language Constructs and Features—Classes and inheritance

General Terms Languages, Theory, Verification

Keywords Separation Logic, Modularity, Classes, Design Patterns

1. Introduction

In the last few years specification and verification of object-oriented programs have seen considerable advances thanks to the introduction of new technologies and tools which are becoming more mature [2, 6, 7, 10, 17, 30].

Despite this remarkable progress, real-world applications still present challenging problems for the verification world. A notorious example is given by design patterns [11], which are largely used in practice, and yet, many of their intricate idioms are still far beyond the reach of the current state of the art [16].

Some key challenges in verifying object-oriented programs are:

Properties across multiple objects We need to be able to express properties about several interacting objects from many different classes.

Call-backs Methods often make calls that in turn will call the original object. This schema can cause great difficulty when one tries to verify it using class/object invariant based approaches.

Modular verification Verification must deal with a single class in isolation. Adding new classes cannot invalidate the verification of pre-existing classes, and changing implementations while preserving specification should only require the re-verification of the changed code.

In recent work, Parkinson and Bierman [25,26] proposed a modular verification technique that addresses these problems. Although theoretically useful, this work has been criticised for not demonstrating its practical utility by providing an automated tool [18, 28]. In this paper we address this criticism, by automating these novel theoretical foundations, and hence bringing them closer to being applicable to real programs. We have extended the abstraction techniques developed for separation logic [9] and married them with these theoretical foundations. The resulting combination has given encouraging practical results in the verification of four popular design patterns.

Contributions. The contributions of this paper can be summarized as follows.

1. An automatic verification tool based on separation logic aiming at object-oriented programs written in Java. The tool, called jStar, integrates two essential parts:
   - A (general) theorem prover for separation logic tailored to object-oriented verification.
   - A (general) symbolic execution and abstraction technique for separation logic tailored to object-oriented verification. With the help of our theorem prover, the abstract interpretation is able to perform fixed-point computation on strong properties resulting by the combination of *heap information* as well as *data contents*. The loop invariant is guessed automatically, minimizing the burden of verification.
2. We bring succinct separation logic specification to the world of automatic object-oriented verification. Pre/post specs in our specification language are simple. Even for intricate examples, such us the observer and visitor patterns, which involves properties of complex heap-allocated objects, the pre/post are straightforward.
3. We provide experimental evidence of the effectiveness of our approach by the automatic verification of four popular design patterns (visitor, subject/observer, factory and pooling). These patterns together are a serious challenge for any other state of the art object-oriented verification technique because of their intense use of aliased global state.

OOPSLA'08, October 19–23, 2008, Nashville, Tennessee, USA.

The paper is organized as follows. We start by introducing jStar with a series of examples, §2. We begin with an easy illustrative example and then present how we have verified the design patterns. We then give details on the architecture of jStar §3, and then introduce the theory behind the tool: the theorem prover §4, the symbolic execution §5, and the abstraction techniques §6. We give an overview of related work, §7, and future work §8. Finally, we conclude in §9.

2. jStar by Example

In this section we demonstrate the verification of a few simple programs and several design patterns.

2.1 Cell/Recell — Inheritance

We begin by presenting a simple example of inheritance to illustrate the key concepts behind jStar. Consider two classes Cell and Recell:

```
class Cell {                    class Recell extends Cell {
    int val;                        int bak;

    void set(int x) {               void set(int x) {
        val=x;                          bak=super.get(); super.set(x);
    }                               }

    int get() {                     int get() {
        return val;                     return super.get();
    }                               }
}                               }
```

The Cell class has a val field, which is updated by the set method and its value is returned by the get method. The subclass Recell has an additional field bak, which stores the previous value the object was set with.

To specify the Cell class, we use a property Val, which describes a Cell's contents. We define the following in the Cell's specification:

$$\mathbf{define}\ Val(x, \{\text{content=}\ y\}) = \mathbf{true} \mid x.val \mapsto y\ ;$$

NOTATION 1. *This definition introduces our formulae, which are divided into two parts $\Pi|\Sigma$. The Π part concerns facts about stack variables and the Σ part about heap allocated objects. A formal definition will be given in §4. Notice that Val does not place any constraint on stack variables. Henceforth, we simply omit the Π part when it is simply* **true** *as in the definition above.*

The above definition defines $Val(x, \{\text{content=}\ y\})$ to mean that the field val of object x has contents y, provided x is precisely of dynamic type Cell. If x is not of this type, then this definition does not constrain the meaning of the property, called *abstract predicate family* [25]. One can view this like a method definition: a method definition specifies the behaviour of a method for a single class, not for all classes.

This definition also defines $Val\$\text{Cell}(x, \{\text{content=}\ y\})$, which can be used by any of Cell's subclasses. It is independent of the actual dynamic type of x, that is, it always asserts x's val field contains y no mater the type of x. We refer to this second property as the *internal property* as it corresponds precisely to the body of the definition for a particular class. Formally, we have the following two axioms:

$$\text{type}(x, \text{Cell}) \implies (\ Val(x, \{\text{content=}\ y\}) \iff Val\$\text{Cell}(x, \{\text{content=}\ y\}))$$

$$Val\$\text{Cell}(x, \{\text{content=}\ y\}) \iff \mathbf{true} \mid x.val \mapsto y$$

where $\text{type}(x, \text{Cell})$ means x is precisely of dynamic type Cell. The first relates the internal property to the general property. Note that, this does not specify the meaning of $Val(x, \{\text{content=}\ y\})$ if x is not of dynamic type Cell. The second specifies the internal property. This can be used to verify the class, but is not available when verifying other classes. That is, other classes (including subclasses) must be independent of the specific internal definition of the predicate, but may mention the predicate abstractly.

For each method, we provide two types of specifications: *static* and *dynamic* [7, 26]. The static specification is used to give a precise specification of the code's behaviour, while the dynamic is used to give a more abstract view of how the method behaves. More specifically, the static specification is used for **super** and private calls and for verifying it is sound to inherit a method. The dynamic specification is used for dynamic dispatch and hence must be implemented by all subclasses (behavioural subtyping [20]).

We can now use these notions to specify the behaviour of the get method:

int get():
static
  pre: $\{\mid \mathbf{this}.val \mapsto X\}$
  post: $\{X = \mathbf{return} \mid \mathbf{this}.val \mapsto X\}$;
dynamic
  pre: $\{\mid\ Val(\mathbf{this}, \{\text{content=}\ X\})\}$
  post:$\{X = \mathbf{return} \mid\ Val(\mathbf{this}, \{\text{content=}\ X\})\}$;

The first specification, the static specification, describes precisely how the method updates the val field.[1] It has the precondition

$$\mid \mathbf{this}.val \mapsto X$$

that specifies that the val field of $\mathbf{this}$ has the value X (a logical variable[2]), and the postcondition

$$X = \mathbf{return} \mid \mathbf{this}.val \mapsto X$$

specifies that the field still has the same value, and this value is returned, $X = \mathbf{return}$.

The second specification, the dynamic specification, describes how the method alters the more abstract Val property, rather than the concrete fields. This enables subclasses to satisfy this specification while changing the concrete behaviour, that is, modifying different fields.

NOTATION 2. *For the sake of brevity, in the following we omit the keywords "pre" and "post" since it is clear that the first formula is the precondition and the second is the postcondition. Moreover, in the following, we only explicitly indicate the* **static** *qualifier, and it should be tacitly clear that the spec is dynamic when the* **static** *qualifiers is not indicated.*

We can make the static specification reveal less implementation details by using the $Val\$\text{Cell}$ property.

int get() **static**:
$\{\mid Val\$\text{Cell}(\mathbf{this}, \{\text{content=}\ \hat{X}\})\ \}$
$\{\ \hat{X} = \mathbf{return} \mid\ Val\$\text{Cell}(\mathbf{this}, \{\text{content=}\ \hat{X}\})\ \}$;

This describes for all subclasses precisely what this method body does, but it does not reveal the fields that are actually modified.

As this pattern of specification is common, we provide a shorthand, which defines the dynamic specification given above, and the second static specification, with the single specification.

int get():
$\{\mid Val\$(\mathbf{this}, \{\text{content=}\ \hat{X}\})\ \}$
$\{\ \hat{X} = \mathbf{return} \mid\ Val\$(\mathbf{this}, \{\text{content=}\ \hat{X}\})\ \}$;

By post-fixing a \$ onto a property it means interpret as the standard property in the dynamic specification, that is without the \$, and as the internal property for the current class in the static specification, in this case $Val\$\text{Cell}$.

[1] In this case, the update of the method get() is the identity function.

[2] Sometimes called auxiliary or ghost variable.

Hence, we can provide both specifications for set with:

void set(**int** x):
{ | $Val\$(\mathbf{this}, \{\text{content}= \hat{X}\})$ }
{ | $Val\$(\mathbf{this}, \{\text{content}= \text{x}\})$ };

We must also specify the behaviour of the constructor:

void <init>():
{ | } { | $Val\$(\mathbf{this}, \{\text{content}= \hat{X}\})$ }

This specification stipulates that constructing an object gives $Val(\mathbf{this}, \{\text{content}= \hat{X}\})$, and from a subclass's **super** constructor call results in the internal property $Val\$\text{Cell}(\mathbf{this}, \{\text{content}= \hat{X}\})$. This enables subclasses to use this property without knowing its meaning.

Now, we turn our attention to the Recell subclass. We must now define what the Val property means for the Recell class.

$\mathbf{define}\ Val(x, \{\text{content}= y;\ \text{old}= z\}) =$
$\quad | \ Val\$\text{Cell}(x, \{\text{content}= y\}) * x.bak \mapsto z$;

If x is of type Recell, this defines $Val(x, \{\text{content}= y;\ \text{old}= z\})$ as the property associated to Val from the superclass Cell and the additional bak field has value z. As we have defined the property with an additional labelled parameter, old, we must also provide a meaning to the property with only the content parameter. In effect, we have width subtyping on the labelled parameters, where missing parameters are existentially quantified. Hence, this defines $Val(x, \{\text{content}= y\})$ for the Recell class as

$$Val\$\text{Cell}(x, \{\text{content}= y\}) * x.bak \mapsto \hat{z}$$

where $\hat{z}$ is an existentially quantified variable. Importantly, by containing the $Val\$$Cell property it allows the Recell to make super calls to the Cell's methods, and also inherit methods, as the precondition of $Val\$$Cell can be provided for the calls.

We can give the specifications for Recell directly.

void <init>(): { | } { | $Val\$(x, \{\text{content}= y;\ \text{old}= z\})$ };

int get():
{ | $Val\$(\mathbf{this}, \{\text{content}= \hat{X};\ \text{old}= \hat{Y}\})$ }
{ $\hat{X} = \mathbf{return}$ | $Val\$(\mathbf{this}, \{\text{content}= \hat{X};\ \text{old}= \hat{Y}\})$ };

void set(**int** x):
{ | $Val\$(\mathbf{this}, \{\text{content}= \hat{X};\ \text{old}= \hat{Y}\})$ }
{ | $Val\$(\mathbf{this}, \{\text{content}= \text{x};\ \text{old}= \hat{X}\})$ };

We must verify that these specifications are valid behavioural subtypes, which follows from width subtyping of labelled parameters. ✯

2.2 Visitor Pattern

Next, we consider the visitor design pattern [11]. We present the source code in Figure 1. Our example involves an abstract syntax tree (Ast) of integer expressions, which are made of constant integer nodes (Const), and plus nodes (Plus) that represent the addition of two integer expressions. The Ast interface requires that each node can "accept" a Visitor.

The Visitor interface has two methods: visitC which is invoked when visiting a Const node; and visitP for visiting Plus nodes. The accept methods in the Const and Plus nodes simply call visitC and visitP respectively.

We consider two concrete visitors in this section: Sum, an operation that calculates the value of the integer expression; and RZ an operation that simplifies the integer expression by removing all the unnecessary additions of zero. The second is complicated by working in-place and swinging pointers where necessary.

We begin by specifying the interfaces Visitor and Ast. We specify these classes with respect to three abstract properties of the state: $Visitor$, Ast and $Visited$. We do not specify what the properties concretely mean as these are only interfaces, and subclasses are free to choose definitions. Intuitively, $Visitor(v, \{\text{context}= \hat{z}\})$ means that v is a visitor ready to visit an expression, and it has some internal data $\hat{z}$; $Visited(v, \{\text{content}= \hat{x};\ \text{context}= \hat{z};\ \text{ast}= a\})$ means that v is a visitor that has visited a tree at a with content $\hat{x}$, and the internal data of the visitor was $\hat{z}$ before it visited this tree. $Ast(a, \{\text{content}= \hat{x}\})$ means a is an integer expression with content $\hat{x}$. We can see these properties as specifying the protocol of the visitor pattern.

We give the interfaces' specifications as

interface Ast
{
void accept(Visitor v):
$\quad$ { | $Visitor(\text{v}, \{\text{context}= \hat{z}\}) * Ast(\mathbf{this}, \{\text{content}= \hat{x}\})$ }
$\quad$ { | $Visited(\text{v}, \{\text{content}= \hat{x};\ \text{context}= \hat{z};\ \text{ast}= \mathbf{this}\})$ };
}
interface Visitor
{
void visitC(Const c):
$\quad$ { type(c, "Const")
$\quad\quad$ | $Visitor(\mathbf{this}, \{\text{context}= \hat{z}\}) * Ast(\text{c}, \{\text{content}= \hat{x}\})$ }
$\quad$ { | $Visited(\mathbf{this}, \{\text{content}= \hat{x};\ \text{context}= \hat{z};\ \text{ast}= \text{c}\})$ };

void visitP(Plus p):
$\quad$ { type(p, "Plus")
$\quad\quad$ | $Visitor(\mathbf{this}, \{\text{context}= \hat{z}\}) * Ast(\text{p}, \{\text{content}= \hat{x}\})$ }
$\quad$ { | $Visited(\mathbf{this}, \{\text{content}= \hat{x};\ \text{context}= \hat{z};\ \text{ast}= \text{p}\})$ };
}

We see that accept will be called on an integer expression with a first parameter that is a visitor, which is ready to visit an integer expression, and when it returns it specifies the visitor must have visited this expression. The specification for visitP and visitC are almost identical except that **this** and the parameter are reversed, and they additionally have type information: for example, type(c, "Const"), which specifies that c is precisely of type Const. This type information captures the double dispatch calling pattern used by the visitor.

To enable us to specify the data associated to the abstract syntax tree, we use two term constructors plus(_,_) and const(_). Using this we can define the Ast property for the Const class as

$\mathbf{export}\ Ast(x, \{\text{content}= y\}) =\ \ y{=}const(\hat{v}) \mid x.v \mapsto \hat{v}$;

This means that x is an integer expression of a constant $\hat{v}$, and the v field contains that value. Here rather than using **define** to specify the property, we use **export** this enables the verification of other classes to use this definition, in particular, the visitors, which depend on the internal representation for efficiency.

Similarly, we define Ast for the Plus class:

$\mathbf{export}\ Ast(x, \{\text{content}= y\}) =\ y{=}plus(\hat{lv}, \hat{rv})$
$\quad | \ x.left \mapsto \hat{l} * Ast(\hat{l}, \{\text{content}= \hat{lv}\})$
$\quad * x.right \mapsto \hat{r} * Ast(\hat{r}, \{\text{content}= \hat{rv}\})$;

This means x is a plus of two integer expressions $\hat{lv}$ and $\hat{rv}$, which are contained in the Asts in the left and right fields respectively.

We specify the accept method for Const as (Plus omitted for brevity)

void accept(Visitor):
$\quad$ {| $Visitor(\text{v}, \{\text{context}= \hat{z}\}) * Ast(\mathbf{this}, \{\text{content}= \hat{x}\})$ }
$\quad$ {| $Visited(\text{v}, \{\text{content}= \hat{x};\ \text{context}= \hat{z};\ \text{ast}= \mathbf{this}\})$ };

void accept(Visitor) **static**:
$\quad$ { type($\mathbf{this}$, "Const") | $Visitor(\text{v}, \{\text{context}= \hat{z}\})$
$\quad\quad * Ast(\mathbf{this}, \{\text{content}= \hat{x}\})$ }
$\quad$ { | $Visited(\text{v}, \{\text{content}= \hat{x};\ \text{context}= \hat{z};\ \text{ast}= \mathbf{this}\})$ };

```
interface Ast{
    public void accept(Visitor v);
}

class Const implements Ast{
    int v;

    Const(int x) {this.v=x;}

    public void accept(Visitor v){
        v.visitC(this);
    }
}

class Plus implements Ast{
    Ast left,right;

    Plus(Ast l, Ast r){
        left=l;
        right=r;
    }

    public void accept(Visitor v){
        v.visitP(this);
    }
}

interface Visitor{
    public void visitC(Const c);
    public void visitP(Plus p);
}

class Sum implements Visitor{
    int amount;
    public void visitP(Plus p){
        p.left.accept(this);
        p.right.accept(this);
    }

    public void visitC(Const c){
        amount+=c.v;
    }
}
```

```
class RZ implements Visitor{
    boolean isZero;
    boolean isChanged;
    Ast newl;

    public void visitC(Const c){
        if(c.v==0)
            this.isZero=true;
    }

    public void visitP(Plus p){
        p.left.accept(this);
        if(this.isZero) {
            this.isChanged=false;
            this.isZero=false;
            p.right.accept(this);
            if(!this.isChanged) {
                this.newl=p.right;
                this.isChanged=true;
            }
        }else{
            if(this.isChanged){
                p.left=this.newl;
                this.isChanged=false;
            }
            p.right.accept(this);
            if(this.isZero){
                this.isChanged=true;
                this.newl=p.left;
                this.isZero=false;
            }else if(isChanged){
                p.right=this.newl;
                this.isChanged=false;
            }
        }
    }
}
```

Figure 1. An example of visitor pattern.

The dynamic specification is identical to the interfaces specification, and the static specification is the same, but with the addition of the dynamic type information. This type information is necessary to verify the double dispatch calling pattern in the visitor pattern. There is a great deal of redundancy in this specification, as it is almost identical to the inherited specification from the interface. The rest of the details of the specification are straightforward and omitted for compactness.

Now we verify the concrete visitors, starting with the Sum visitor. To specify the functional behaviour of this visitor we must define a function that sums the integer expression:

$$\begin{aligned} sum(const(v)) &= v \\ sum(plus(l,r)) &= sum(l) + sum(r) \end{aligned}$$

We can define the $Visitor$ and $Visited$ properties for this class as:

$$\textbf{define } Visitor(x, \{\text{context= } y\}) = \mid x.amount \mapsto y;$$

$$\begin{aligned} &\textbf{define } Visited(x, \{\text{content= } z;\ \text{context= } y;\ \text{ast= } a\}) = \\ &\quad \mid x.amount \mapsto y + sum(z) \\ &\quad * Ast(a, \{\text{content= } z\}); \end{aligned}$$

The $Visitor$ defines that the amount field has some pre-existing value; and the $Visited$ property defines that the amount field contains the addition of the pre-existing value y and the summation of the integer expression z. The $Visited$ property also contains the Ast that it has just read, this is necessary to allow other visitors to update the structure (see the next example visitor). The method specifications is straightforward and therefore omitted.

Now, we turn our attention to the second concrete visitor: RZ. This visitor removes all the additions of zero from an expression. Mathematically we first define a function rz that removes the zeros

$$\begin{aligned} rz(const(x)) &= const(x) \\ rz(plus(x,y)) &= \begin{cases} rz(x) & \text{if } rz(y)=const(0) \\ rz(y) & \text{if } rz(x)=const(0) \\ plus(rz(x), rz(y)) & \text{otherwise} \end{cases} \end{aligned}$$

For example, $rz(plus(const(0), plus(const(1), const(0))))$ is $const(1)$, and $rz(const(0))$ is $const(0)$.

The Visitor implementation is complicated by the in-place nature of the update, that is the visitor does not allocate any new storage nodes in the expression, it simply updates old ones. The $Visitor$ property simply states that the $isZero$ and $isChanged$ fields are both initially set to **false**. The $newl$ field can have any arbitrary value, and this is indicated by the existentially quantified value $\hat{z}$

```
import java.util.*;
import java.sql.*;

class DBPool{
    LinkedList<Connection> conns;
    String url,user,password;

    DBPool(String url,String user, String password){
        conns = new LinkedList<Connection>();
        this.url = url;
        this.user = user;
        this.password = password;
    }

    public Connection getResource() throws SQLException{
        if(conns.size()==0)
            return DriverManager.getConnection(url,user,password);
        return conns.removeFirst();
    }

    public void freeResource(Connection db) throws SQLException{
        if(conns.size() >= 20)
            db.close();
        else
            conns.add(db);
    }
}
```

Figure 2. Database pool example

$$\begin{array}{l}\textbf{define } \mathit{Visitor}(x, \{\text{context=}\, y\}) = | \\ \quad x.\mathit{isZero} \mapsto \textbf{false}() \\ \quad * \; x.\mathit{isChanged} \mapsto \textbf{false}() \\ \quad * \; x.\mathit{newl} \mapsto \hat{z};\end{array}$$

The $\mathit{Visited}$ property is more complex:

$$\begin{array}{l}\textbf{define } \mathit{Visited}(x, \{\text{content=}\, z;\ \text{context=}\, y;\ \text{ast=}\, a\}) = \\ \quad | \; x.\mathit{isZero} \mapsto \hat{iz} \\ \quad * \; x.\mathit{isChanged} \mapsto \hat{ic} \\ \quad * \; x.\mathit{newl} \mapsto \hat{al} \\ \quad * \; (\hat{ic} \neq \textbf{false}() \mid \mathit{Ast}(\hat{al}, \{\text{content=}\, rz(z)\}) \\ \qquad || \; \hat{ic} = \textbf{false}() \mid \mathit{Ast}(a, \{\text{content=}\, rz(z)\})) \\ \quad * \; (\hat{iz} \neq \textbf{false}() \wedge rz(z) = \mathit{const}(\textbf{zero}()) \mid \mathsf{emp} \\ \qquad || \; \hat{iz} = \textbf{false}() \wedge rz(z) \neq \mathit{const}(\textbf{zero}()) \mid \mathsf{emp});\end{array}$$

We use || to mean disjunction, which combines two formula (both a pure and spatial part) to give a spatial formula. We use emp for the empty spatial formula. First, we relate the fields to existential variables, so they can be used in the rest of an assertion. The next part specifies if $\hat{iz}$ is true, then the content $rz(z)$ is zero. If $\hat{iz}$ is false, then it is not zero. Finally, we specify that if $\hat{ic}$ is true, then $\hat{al}$ contains the updated integer expression, otherwise the original node a is the updated integer expression.

The RZ class highlights the potential compactness of the specifications. The code for this class is almost incomprehensible, yet the two properties clearly express how the class works, and that it satisfies the functional specification of rz. ✯

2.3 Connection Pool — Ownership Transfer

Next we present a simple example of a connection pool for a database [12]. This example illustrates ownership transfer [23, 25], that is when a connection is freed with freeResource it should no longer be used. We present the source code in Figure 2.

The class has two methods: getResource and freeResource. The first returns a pointer to an unused database connection. This might have come from its internal cache of connections stored in the linked list pointed to by the conns field, or it could be a freshly allocated one from the database library. The second method, freeResource, either closes the connection it is passed, or adds the connection to its internal list of connections.

Next, we present the specification of these methods:

java.sql.Connection getResource():

$$\begin{array}{l}\quad \{ \mid \mathit{DBPool}\$(\textbf{this}, \{\text{type} = \hat{t}\})\} \\ \quad \{ \mid \mathit{DBPool}\$(\textbf{this}, \{\text{type} = \hat{t}\}) \\ \qquad * \; \mathit{DBConnection}(\textbf{return}, \{\text{connection=}\, \hat{t}\}) \; \};\end{array}$$

void freeResource(java.sql.Connection db):

$$\begin{array}{l}\quad \{ \mid \mathit{DBPool}\$(\textbf{this}, \{\text{type} = \hat{t}\}) \\ \qquad * \; \mathit{DBConnection}(\text{db}, \{\text{connection=}\, \hat{t}\}) \; \} \\ \quad \{ \mid \mathit{DBPool}\$(\textbf{this}, \{\text{type} = \hat{t}\}) \; \};\end{array}$$

The getResource() method's pre-condition specifies we must have a connection pool, DBPool, of some type $\hat{t}$, and the postcondition specifies that we still have this pool, but additional we are returned a connection connected to $\hat{t}$, $\mathit{DBConnection}$. Here $\hat{t}$ is used to represent the database connection strings (url, user and password). Due to the underlying separation logic, we know this connection is separate from any other connection in use.

The specification of freeResource is the converse, its precondition specifies that you must have a pool and a connection, and the postcondition specifies you only have the pool. Hence, once freeing a connection you can no longer use the connection.

We can define the DBPool property as

$$\begin{array}{l}\textbf{define } \mathit{DBPool}(x, \{\text{type=}t\}) = \; t{=}\mathit{sql}(\hat{url}, \hat{user}, \hat{password}) \mid \\ \quad x.\mathit{url} \mapsto \hat{url} * \\ \quad x.\mathit{user} \mapsto \hat{user} * \\ \quad x.\mathit{password} \mapsto \hat{password} * \\ \quad x.\mathit{conns} \mapsto \hat{y} * \\ \quad \mathit{LinkedList}(\hat{y}, \hat{R}) * \mathit{DBSet}(\mathit{setof}(\hat{R}), t) \; ;\end{array}$$

This defines that the fields that store the url, user and $\mathit{password}$ contain the right data, and that the conns fields contains $\hat{y}$, which is a list with contents $\hat{R}$, represented by the property $\mathit{LinkedList}$. The function setof converts a list into a multiset. We use a special predicate DBSet to represent that each value in the list points to a database connection. We discuss the precise details of this predicate later in the paper. ✯

2.4 Factory Pattern

The database connection pool in the previous section could be re-factored to allow greater code reuse. In particular, we could break apart the database specific component from the pool of resources. We can do this re-factoring with either the Factory pattern or the Template method pattern [11]. We have performed and verified both re-factorings. Here we just present the Factory pattern as the code is simpler, and hence so is the specification.

We define a ResourceFactory interface that makes and destructs resources. We can then parametrize a pooling class, ResPool, with a factory to create the objects that will be pooled. Finally, we make a ConnectionFactory class that implements the ResourceFactory interface. We present the interface and the classes in Figure 3.

We give specification to the factory that are very similar to the connection pool from earlier:

interface ResourceFactory
{
Object makeResource():

$$\begin{array}{l}\quad \{ \mid \mathit{ResourceFactory}(\textbf{this}, \{\text{type=}\hat{t}\})\} \\ \quad \{ \mid \mathit{Resource}(\textbf{this}, \{\text{handle=}\, \textbf{return};\ \text{type=}\, \hat{t}\}) \\ \qquad * \; \mathit{ResourceFactory}(\textbf{this}, \{\text{type=}\hat{t}\})\};\end{array}$$

```
import java.util.*;
import java.sql.*;

public interface ResourceFactory<R,E extends Exception>{
    public R makeResource() throws E;
    public void destructResource(R r) throws E;
}

public class ResPool<R,E extends Exception>{
    LinkedList<R> resources;
    ResourceFactory<R,E> rf;

    ResPool(ResourceFactory<R,E> rf){
        resources=new LinkedList<R>();
        this.rf=rf;
    }

    public R getResource() throws E{
        if(resources.size()==0)
            return rf.makeResource();
        return resources.removeFirst();
    }

    public void freeResource(R r) throws E{
        if(resources.size()>=20)
          rf.destructResource(r);
        else resources.add(r);
    }
}

class ConnectionFactory
    implements ResourceFactory<Connection,SQLException>{
    String url, user, password;

    ConnectionFactory(String url, String user, String password){
        this.url=url; this.user=user; this.password=password;
    }

    public Connection makeResource() throws SQLException{
        return DriverManager.getConnection(url,user,password);
    }

    public void destructResource(Connection c) throws SQLException{
        c.close();
    }
}
```

Figure 3. Factory pattern

```
void destructResource(Object r):
        { | Resource(this, {handle= r; type= t̂})
          *ResourceFactory(this, {type=t̂}) }
        { | ResourceFactory(this, {type=t̂}) };
}
```

However, there is one key difference rather than specify precisely what resource will be returned, and consumed, we use a property $Resource(\mathbf{this}, \{\text{handle=}\,\mathbf{return};\ \text{type=}\,\hat{t}\})$ that is dependent on the particular implementation of the ResourceFactory. This enables each implementation to specify what $Resource$ means, for example, one class specifies it as a database connection and another as an XSLT processor.

We can then specify the ResPool class as

```
class ResPool {
define ResPool(x, {factory= f; type= t}) = |
    x.resources ↦ ŷ *
    x.rf ↦ f *
    ResourceFactory(f, {type=t}) *
    LinkedList(ŷ, R̂) * IterRes(setof(R̂), f, t) ;

void <init>(ResourceFactory rf):
    { |ResourceFactory(rf, {type=t̂}) }
    { |ResPool$(this, {factory= rf; type= t̂}) };

void freeResource(java.lang.Object r):
        { | ResPool$(this, {factory= f̂; type= t̂})
          * Resource(f̂, {handle= r; type= t̂}) }
        { | ResPool$(this, {factory= f̂; type= t̂}) };

java.lang.Object getResource():
        { | ResPool$(this, {factory= f̂; type= t̂}) }
        { | ResPool$(this, {factory= f̂; type= t̂})
          * Resource(f̂, {handle= return; type= t̂}) };
}
```

Here, we must use a new predicate $IterRes(X, f, t)$ to mean that there exists a $Resource(f, \{\text{handle=}\,i;\ \text{type=}\,t\})$ predicate for each element i of the set X. In separation logic, this would be given with the iterated separating conjunction:

$$\circledast_{i \in X}\, Resource(f, \{\text{handle=}\,i;\ \text{type=}\,t\})$$

The definition of $ResPool$ means that for each element of the list from $\hat{y}$, we have a $Resource$ predicate, that was created by the $ResourceFactory\ f$.

Finally, in the ConnectionFactory specification we define the $ResourceFactory$ and $Resource$ properties:

```
define ResourceFactory(x, {type=t}) =
    t = sql(url̂, usêr, passŵord) | x.url ↦ url̂ *
    x.user ↦ usêr * x.password ↦ passŵord;

export Resource(x, {handle= y; type= t}) = |
      DBConnection(y, {connection= t});
```

The most important part of this specification is that the $Resource$ definition is **export**ed. This means that, since $Resource$ is defined in the ConnectionFactory specification then any class can use the following fact in its verification:

$$\text{type}(f, \text{"ConnectionFactory"}) \mid Resource(f, \{\text{handle=}\,x;\ \text{type=}\,t\}) \implies DBConnection(x, \{\text{type} = t\})$$

and the reverse

$$\text{type}(f, \text{"ConnectionFactory"}) \mid DBConnection(x, \{\text{type=}t\}) \implies Resource(f, \{\text{handle=}\,x;\ \text{type=}\,t\})$$

This means that any client of the resource pool, who knows the type of the internal factory is a ConnectionFactory, will know it allocates and deallocates database connections.

We omit the method specifications as they follow directly from the interface. ✯

2.5 Subject/Observer

We conclude our illustration of jStar by demonstrating an example of the subject/observer pattern [11]. This has been a popular design pattern for challenging recent verification techniques [1, 3, 15, 18]. We present an example subject/observer pattern in Figure 4. We explain the code through its specification.

We can specify the properties of the IntegerList as:

```
define Subject(s, {obs= O; vals= V}) =
 | SubjectInternal$IntegerList(s, {obs= O}) *
     SubjectData(s, {vals= V}) ;

define SubjectInternal(s, {obs= O}) =
 | s.observers ↦ ô * LinkedList(ô, O) ;

export SubjectData(s, {vals= V}) =
 | s.list ↦ l̂ * LinkedList(l̂, V) ;
```

```
public interface Subject{
        public void addObserver(Observer o);
        public void removeObserver(Observer o);
}

public interface Observer{
        public void update(Subject o);
}

public class IntegerSize implements Observer{
   IntegerList bag;
   int size;

   public IntegerSize(IntegerList bag){
      this.bag=bag;
      bag.addObserver(this);
   }

   public void update(Subject o){
      if(o==bag)
          size=bag.list.size();
   }
}
```

```
public class IntegerList implements Subject{

   LinkedList list=new LinkedList();
   LinkedList observers=new LinkedList();

   public void beginModification(){}
   public void endModification(){notifyObservers();}

   public void addObserver(Observer o){
      observers.add(o);
      o.update(this);
   }

   public void removeObserver(Observer o){
      observers.remove(o);
   }

   private void notifyObservers() {
      Iterator i=observers.iterator();
      while(i.hasNext()) {
         Observer o=(Observer)i.next();
         o.update(this);
      }
   }
}
```

Figure 4. Source code for Subject/Observer example

$$\mathbf{export} SubjectObs(s, \{\text{obs}= O;\ \text{vals}= V\}) =$$
$$\mid Subject\$\text{IntegerList}(s, \{\text{obs}= O;\ \text{vals}= V\})$$
$$* \ ObsSet(O, V, s);$$

We define the $Subject$ property as being composed of two parts the data, $SubjectData$, and the internal state, $SubjectInternal$. The latter represents the internal structures for keeping the list of Observers, and the former is the data associate to the subject, in this case a list of Integers. We additionally provide a predicate to represent the aggregate structure of the set of Observers and the subject: $SubjectObs$. This property is **export**ed, so that clients can access the individual observers as well as the whole aggregate.

We also **export** the $SubjectData$ so that clients can manipulate the state associated to the IntegerList using the beginModification and endModification methods. The client should call beginModification to gain access to the internal data, and upon completion of the modification should call endModification.

void beginModification():

$$\{ \mid SubjectObs\$(\mathbf{this}, \{\text{obs}= \hat{O};\ \text{vals}= \hat{V}\}) \}$$
$$\{ \mid SubjectInternal\$(\mathbf{this}, \{\text{obs}= \hat{O}\})$$
$$* \ SubjectData(\mathbf{this}, \{\text{vals}= \hat{V}\}) * ObsSet(\hat{O}, \hat{V}, \mathbf{this}) \};$$

void endModification():

$$\{ \mid SubjectInternal\$(\mathbf{this}, \{\text{obs}= \hat{O}\})$$
$$* \ SubjectData(\mathbf{this}, \{\text{vals}= \hat{V}_2\}) * ObsSet(\hat{O}, \hat{V}, \mathbf{this}) \}$$
$$\{ \mid SubjectObs\$(\mathbf{this}, \{\text{obs}= \hat{O};\ \text{vals}= \hat{V}_2\}) \};$$

The specification of beginModification means that we can call the method if we have the property of the aggregate structure, and after the call we have the structure broken into parts, including the $SubjectData$ property, which allows the client to modify the associated data. The specification of endModification takes a $SubjectData$ predicate with a potentially different value, $\hat{V}_2$ to the observers, and makes the observers consistent with the new value. Note, if we had **export**ed the $Subject$ property as well, then for this example the beginModification method could be removed.

We provide methods for adding and removing observers from the aggregate structure:

void addObserver(Observer o):

$$\{ \mid SubjectObs\$(\mathbf{this}, \{\text{obs}= \hat{O};\ \text{vals}= \hat{V}\})$$
$$* \ Observer(\text{o}, \{\text{vals}= \hat{v_2};\ \text{subject}= \mathbf{this}\}) \}$$
$$\{ \mid SubjectObs\$(\mathbf{this}, \{\text{obs}= add(\text{o}, \hat{O});\ \text{vals}= \hat{V}\}) \};$$

void removeObserver(Observer o):

$$\{ \mid SubjectObs\$(\mathbf{this}, \{\text{obs}= add(\text{o}, \hat{O});\ \text{vals}= \hat{V}\}) \}$$
$$\{ \mid SubjectObs\$(\mathbf{this}, \{\text{obs}= \hat{O};\ \text{vals}= \hat{V}\})$$
$$* \ Observer(\text{o}, \{\text{vals}= \hat{v_2};\ \text{subject}= \hat{s}\}) \};$$

The first simply takes the aggregate and an $Observer$, and puts it into the aggregate, and the latter removes an observer from the aggregate.

Finally, we have a method notifyObservers, which when given a $Subject$ and an out of date set of observers, $ObsSet$, will update the set of observers to the correct value:

void notifyObservers():

$$\{ \mid Subject\$(\mathbf{this}, \{\text{obs}= \hat{O};\ \text{vals}= \hat{V}\}) * ObsSet(\hat{O}, \hat{V}_2, \mathbf{this}) \}$$
$$\{ \mid Subject\$(\mathbf{this}, \{\text{obs}= \hat{O};\ \text{vals}= \hat{V}\}) * ObsSet(\hat{O}, \hat{V}, \mathbf{this}) \};$$

The behaviour of notifyObservers depends on the specification of the Observer interface:

interface Observer {
void update(Subject s):

$$\{ \mid Observer(\mathbf{this}, \{\text{vals}= \hat{v};\ \text{subject}= \text{s}\})$$
$$* \ SubjectData(\text{s}, \{\text{vals}= \hat{v_2}\}) \}$$
$$\{ \mid Observer(\mathbf{this}, \{\text{vals}= \hat{v_2};\ \text{subject}= \text{s}\})$$
$$* \ SubjectData(\text{s}, \{\text{vals}= \hat{v_2}) \};$$

}

This says the $Observer$ will correctly update itself to the value current in $SubjectData$.

Finally, we turn our attention to the constructors specification:

void <init>():

$$\{ \mid \}$$
$$\{ \mid Subject(s, \{\text{obs}= empty();\ \text{vals}= empty()\}) \};$$

void <init>() **static**:

$$\{ \mid \}$$
$$\{ \mid SubjectData\$\text{IntegerList}(s, \{\text{vals}= empty()\})$$
$$* \ SubjectInternal\$\text{IntegerList}(s, \{\text{obs}= empty()\}) \};$$

Pattern	Time(s)	LOC	Result
Visitor	0.40	71	Yes
Connection Pool	0.06	27	Yes
Factory Pattern	0.10	46	Yes
Subject/Observer	0.50	49	Yes

Table 1. Experimental results of the example patterns performed with jStar on a 1.66 GHz Intel Core Duo, 2 GB Ram.

We might have expected the **static** post-condition of the constructor to be $Subject\$\text{IntegerList}(s, \{\text{obs}= empty();\ \text{vals}= empty()\})$ but unfortunately, this is not true. The $Subject$ predicate for this class contains the $SubjectData$ property, for the precise type of the object being constructed. However, as this constructor is potentially inherited we do not know the object is precisely of type IntegerList: it could be a subtype. Hence, we can only provide this weaker specification. However, this allows subclasses to change the $SubjectData$ predicate and still inherit the code from this classes methods: the verification of all the instance methods of class is abstract in the definition of $SubjectData$. ✯

2.6 Experimental Results on Patterns Verification.

We have run jStar on all the pattern examples from Section 2. Table 1 reports the results. All static and dynamic specs for every example were automatically verified. The Result column shows that all patterns meet their specifications. ✯

3. The jStar Architecture

Next, we give a global overview of jStar's internal structure, depicted in Figure 5. A detailed description of each component will be given in the next section of this paper. jStar is composed by two main components: a theorem prover and the symbolic execution module. The prover is called by the symbolic execution during the verification process to decide implications or to perform frame inference. The symbolic execution module is responsible for the fixed point computation of invariants.

jStar accepts programs written in Jimple, which is one of the Soot toolkit intermediate representations [29] designed to analyze Java programs. Hence, we use Soot for parsing Java into Jimple, the latter is then parsed by jStar into its internal data structures. jStar is implemented in OCaml.

Other input files are used by jStar for the verification of a Java program: (1) pre/post condition specifications of the program's methods as well as the specification of the methods it calls; (2) the logic rules — i.e., the theory — used by the theorem prover to decide entailment and other implications; and (3) the abstraction rules, which are special rules used to ensure convergence in the fixed-point computation of loop-invariants. Abstraction rules are an extension of the logical rules for deciding implications. ✯

4. Theorem Prover

Next, we describe our abstract theorem prover for separation logic. The design is based on the entailment checker in smallfoot [4].

4.1 Formulae

Let Var be a countable set of program variables (ranged over by $x, y, \ldots$) and $\hat{Var}$ a countable set of existential variables (ranged over by $\hat{x}, \hat{y}, \ldots$). A formula, H, is a restricted form of separation

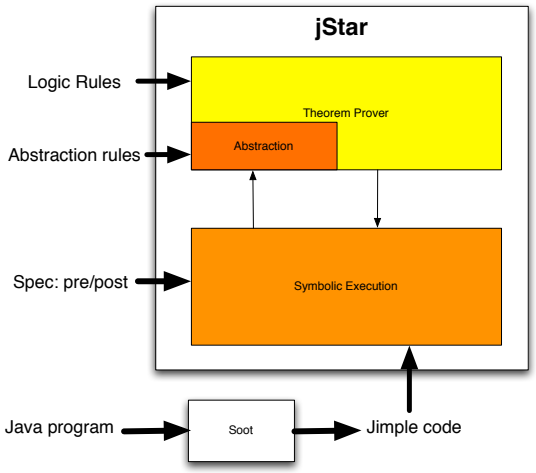

Figure 5. jStar architecture

logic formula, defined by the following grammar.

$$
\begin{array}{lcll}
E & ::= & x \mid \hat{x} \mid \mathsf{nil} \mid \ldots & \text{(Expressions)} \\
P & ::= & E = F \mid E \neq F \mid p(\overline{E}) & \text{(Pure predicates)} \\
S & ::= & s(\overline{E}) & \text{(Spatial predicates)} \\
\Pi & ::= & \mathsf{true} \mid P \wedge \Pi & \text{(Pure part)} \\
\Sigma & ::= & \mathsf{emp} \mid S * \Sigma & \text{(Spatial part)} \\
H & ::= & \Pi \wedge \Sigma & \text{(Formula)}
\end{array}
$$

Note that we use the letter H for formulae and in the following sections we use formulae to represent *symbolic heaps*. Given a formula $H = \Pi \wedge \Sigma$, we call Π the *pure part*, whereas Σ is called the *spatial part*. We denote by $Heaps$ the set of all symbolic heaps and we use hatted variables to (implicitly) denote existentially quantified variables. That is, $\Pi \wedge \Sigma$ is a shorthand for

$$\exists \hat{x}_1, \ldots, \hat{x}_n . \Pi \wedge \Sigma$$

where $\hat{x}_1, \ldots, \hat{x}_n$ are the existential variables occurring in $\Pi \wedge \Sigma$.

The pure part Π is a conjunction of *pure predicates* which states facts about the stack variables and existential variables (e.g., $x = \mathsf{nil}$), but are not concerned with heap allocated objects. The spatial part is the $*$ conjunction of spatial predicates, i.e., related to heap facts. In separation logic, the formula

$$S_1 * S_2$$

holds in a heap that can be split into two *disjoint parts* where in one of them the *only* allocated memory is described by S_1 and in the other only by S_2.

We use a *field splitting model*, i.e., in our model, objects are considered to be a compound entities composed by fields which can be split by $*$.[3] Notice that if S_1 and S_2 describe the same field of an object than $S_1 * S_2$ implies false.

The predicate emp says that there is nothing allocated in the heap. The prover allows the definition of arbitrary pure predicates p and spatial predicates s.

Here it is worth to mention a fundamental rule which gives the bases of local reasoning in separation logic:

$$\frac{\{H_1\}\ C\ \{H_2\}}{\{H_1 * H\}\ C\ \{H_2 * H\}}\ \text{Frame Rule}$$

[3] An alternative model would consider the granularity of $*$ at the level of objects. In that case, objects cannot be split by $*$ since they are the smallest unit in the heap.

where C does not assign to H's free variables [22]. The frame rule allows us to circumscribe the region of the heap which is touched by C, (in this case H_1), perform local surgery, and combine the result with the frame, i.e. the part of the heap not affected by the command C (in this case H). In the rest of the paper we make intensive, although often tacit, use of the frame rule. ✯

4.2 Proof Rules

Our prover works on sequents of the form

$$\Sigma_f \mid \Pi_1 \mid \Sigma_1 \vdash \Pi_2 \mid \Sigma_2$$

We call $\Pi_1 \mid \Sigma_1$ the assumed formula, $\Pi_2 \mid \Sigma_2$ the goal formula, and Σ_f the subtracted (spatial) formula. The semantics of a judgement are:

$$\Pi_1 \wedge (\Sigma_f * \Sigma_1) \Longrightarrow \Pi_2 \wedge (\Sigma_f * \Sigma_2)$$

The subtracted formula, Σ_f, is used to allow predicates to be removed from both sides without losing information. This makes finding complete proof rules easier, while guaranteeing termination.

The prover has built in simplification rules. We present just two here:

$$\frac{\Sigma_f * S \mid \Pi_1 \mid \Sigma_1 \vdash \Pi_2 \mid \Sigma_2}{\Sigma_f \mid \Pi_1 \mid \Sigma_1 * S \vdash \Pi_2 \mid \Sigma_2 * S}$$

$$\frac{\Sigma_f[E/x] \mid \Pi_1[E/x] \mid \Sigma_1[E/x] \vdash \Pi_2[E/x] \mid \Sigma_2[E/x]}{\Sigma_f \mid \Pi_1 \wedge x = E \mid \Sigma_1 \vdash \Pi_2 \mid \Sigma_2}$$

These rules are used to prove the implications, but can also be supplemented by user supplied rules, for example:

rule field_remove1:
| | field(?e1,?e2,?e3) ⊢ | field(?e1,?e2,?e4)
if
field(?e1,?e2,?e3) | | ⊢ ?e3=?e4 |

The empty parts of the sequent are simply preserved, and the ?e1, ?e3, ?e2 and ?e4 are variables that can be unified with any expression (as opposed to _e, which can only be unified with existential variables[4]). The definition is equivalent to the following rule:

$$\frac{\Sigma_f * \text{field}(E_1, E_2, E_3) \mid \Pi_1 \mid \Sigma_1 \vdash \Pi_2 \wedge E_3 = E_4 \mid \Sigma_2}{\Sigma_f \mid \Pi_1 \mid \Sigma_1 * \text{field}(E_1, E_2, E_3) \vdash \Pi_2 \mid \Sigma_2 * \text{field}(E_1, E_2, E_4)}$$

This can be read as saying, if you have the same field on either side of the entailment, then they must have the same value.

We can specify a property is true of either the subtracted or assumed formula by placing it on the far left hand side of a sequent.

rule field_field_contradiction :
field(?e1,?e2,?e3) * field(?e1,?e2,?e4) | | ⊢ |
if

This rule is actually equivalent to

$$\frac{\text{field}(E_1, E_2, E_3) * \text{field}(E_1, E_2, E_4) \in \Sigma_f * \Sigma_1}{\Sigma_f \mid \Pi_1 \mid \Sigma_1 \vdash \Pi_2 \mid \Sigma_2}$$

The rule means if the same field is contained in either of the subtracted or assumed formula, then we have assumed a contradiction (because of the property of $*$ in separation logic as discussed in Section 4.1) and the proof is complete. The user of the theorem prover can instantiate our framework with any collection of these rules, by giving in input the appropriate Logic Rules file.

[4] In all the jStar's input files existentially quantified variables are prefixed with an underscore.

The prover simply uses these rules for proof search. For entailment checking the following basic axiom is used.

$$\frac{}{\Sigma_f \mid \Pi_1 \mid \mathsf{emp} \vdash \mathsf{true} \mid \mathsf{emp}}$$

As an example let us describe some of the rules used for the Connection Pool pattern in Section 2.3. There, we described a $DBSet(S,t)$ predicate, which represented a set of connections S with parameter t. In separation logic, we would represent this as

$$\circledast_{i \in S} DBConnection(i,t)$$

We can encode this into the prover as follows. We represent sets with three functions $add(x,S)$, $union(S_1,S_2)$ and $empty()$, which mean add x to the set S, combine the sets S_1 and S_2, and the empty set, respectively. We then encode $DBSet$ as follows:

rule dbsetleft_add:
| | DBSet(add(?x,?y),?t) ⊢ |
if
| | DBSet(?y,?t) * DBConnection(?x,{connection = ?t}) ⊢ |

rule dbsetleft_union:
| | DBSet(union(?x,?y),?t) ⊢ |
if
| | DBSet(?y,?t) * DBSet(?x,?t) ⊢ |

rule dbsetleft_empty:
| | DBSet(empty(),?t) ⊢ |
if
| | ⊢ |

We show only the left hand rules, and have a similar three right hand rules.

Finally, we present a rule that allows us to try to unify these more complex terms

rule DBSet :
| | DBSet(?x,?y) ⊢ | DBSet(?z,?y)
without
?x!=?z
if
DBSet(?x,?y) | | ⊢ ?x=?z |
or
| | DBSet(?x,?y) ⊢ ?x!=?z | DBSet(?z,?y)

Here without ?x!=?z means do not apply this rule if the inequality is present in the sequent. This ensures we only apply this rule once. The or means try to prove the first premise, and if that fails try the second, hence this rule demonstrates the backtracking of the theorem prover. It says try to prove ?x=?z, and if you fail try something else. This rule is incomplete, and hence other similar rules for add, $union$ and $empty$ are added.

The prover also can be extended with *rewrite rules*. The sum definition from earlier is converted to:

rewrite sum_plus :
sum(plus(?x,?y)) = sum(?x) + sum(?y)

rewrite sum_const :
sum(const(?n)) = ?n

These rewrite rules are used to simplify terms.

Frame Inference. A key part of symbolic execution requires frame inference, that is given two formula (heaps) H_1 and H_2 find a third H_3 such that

$$H_1 \Longrightarrow H_2 * H_3 \ .$$

As with smallfoot, we can find these by simply altering the basic axiom to

$$\frac{\mathsf{addToFrame}(\Pi_1 \mid \Sigma_1)}{\Sigma_f \mid \Pi_1 \mid \Sigma_1 \vdash \mathsf{true} \mid \mathsf{emp}}$$

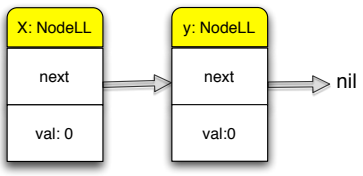

Figure 6. The heap described in Example 1.

This collects all the leftover formulae, $\Pi_1 \mid \Sigma_1$, from the proof, and by taking the disjunction of these formulae forms the frame H_3. Consider the following example inference

$$\frac{\frac{\mathsf{field}(X, \texttt{"val"}, Z) \mid \hat{v} = Z \mid \mathsf{field}(X, \texttt{"bak"}, W) \vdash \mathsf{true} \mid \mathsf{emp}}{\mathsf{field}(X, \texttt{"val"}, Z) \mid \mathsf{true} \mid \mathsf{field}(X, \texttt{"bak"}, W) \vdash \hat{v} = Z \mid \mathsf{emp}}}{\mathsf{emp} \mid \mathsf{true} \mid \mathsf{field}(X, \texttt{"val"}, Z) * \mathsf{field}(X, \texttt{"bak"}, W) \vdash \mathsf{true} \mid \mathsf{field}(X, \texttt{"val"}, \hat{v})}$$

Here we find the frame to be $\hat{v} = Z \mid \mathsf{field}(X, "bak", W)$. This says we have the bak field spare, and the existential variable $\hat{v}$ has been unified with Z. ✯

5. Symbolic Execution

Next, we define our symbolic execution for object-oriented programs taking inspiration from [5, 9]. A symbolic execution defines the effect of a Jimple command on a symbolic state. Symbolic states are specified in terms of separation logic formulae.

5.1 Symbolic Heaps

Let $FNames$, $CNames$, $TNames$ and $MNames$ be a countable set of field, class, type and method names respectively. A *signature* of an object field/method is a triple

$$\langle C\colon t\ f\rangle \in CNames \times TNames \times (FNames \cup MNames)$$

indicating that the field f in objects of class C has type t. In the following we indicate by Sig the set of all signatures and by $\rho, \rho_1, \ldots$ its elements.

The predicate $x.\langle C\colon t\ f\rangle \mapsto E$ states that the object denoted by x *points to* the value E by the field f.

In the examples section, we omitted the class and type parameters for clarity, but they are essential in the symbolic execution for dealing with field shadowing.

EXAMPLE 1. *The symbolic heap*

$$x.\langle NodeLL\colon NodeLL\ next\rangle \mapsto y * x.\langle NodeLL\colon int\ val\rangle \mapsto 0 * \\ y.\langle NodeLL\colon NodeLL\ next\rangle \mapsto \mathsf{nil} * y.\langle NodeLL\colon int\ val\rangle \mapsto 0$$

describes a heap where there are precisely two allocated objects (of class NodeLL) x and y linked by the next field and whose value is initialized to 0. Figure 6 shows a pictorial view of this heap. The definition of class NodeLL is

```
class NodeLL {
    int val;
    NodeLL next;
}
```

✯

5.2 Rules for Symbolic Execution.

Symbolic execution implements the function:

$$\mathsf{exec} : Stmts \times Heaps \to \mathcal{P}(Heaps) \cup \{\top\}$$

which takes a Jimple statement and a heap and returns a set of resulting heaps after the execution of the statement or the special element $\top$ indicating that there is a possible error. Table 2 defines the transformations for basic commands which implement exec.

The rule Assignment 1, when executed in a state H adds the information that in the resulting state x is equal to E. As in standard Hoare/Floyd style assignment, all the occurrences of x in H, and E are replaced by a fresh existential quantified variable $\hat{x}$. The Mutation rule updates (in-place) the value of the field f of object x with value E_2. The Look-up rule adds an equality between x and the content of the field f of object E to the resulting state. As for Assignment 1, the occurrences of x in the input state is replaced by a fresh existential variable $\hat{x}$ in the output state. The Return rule assigns the value to be returned to the special variable ret which will be replaced when the control flow exits the method. The rule Invoke deals with method invocation. The resulting state is determined by the function jsr which is applied to the specs of the method with signature $\langle C\colon t\ m\rangle$. In Jimple there are two main kinds of invocation: *instance invoke* and *class invoke*. Here, we focus on instance invokes as the class invokes are trivial.[5] An instance of invoke can be either `specialinvoke` or `virtualinvoke`.[6] We use invoke to range over these two possibilities. Jimple translates **super** and **private** calls into `specialinvoke`, and dynamically dispatched calls into `virtualinvoke`. We define the following indexed function for specifications:

$$\mathsf{spec}_{\mathsf{invoke}} : Sig \to \mathcal{P}(Specs)$$

where $Specs$ is the set of all pre/post specification. When invoke is `virtualinvoke` the function gives all the dynamic specifications associated to a method, and for `specialinvoke` it gives all the static specifications. Note that Invoke rule is non-deterministic. The function

$$\mathsf{jsr} : Specs \times Heaps \times Var^* \to Heaps$$

is defined as

$$\mathsf{jsr}(\{P\}m(\vec{w})\{Q\}, H, \vec{v}) = \begin{cases} H' * Q[\vec{v}/\vec{w}] & \text{if } H \vdash P[\vec{v}/\vec{w}] * H' \\ \top & \text{otherwise} \end{cases}$$

Given a pre/post spec $\{P\}m(\vec{w})\{Q\}$, the current heap H where the m is called, and the actual parameters $\vec{v}$, jsr invokes the theorem prover for the following *frame inference* question:

$$H \vdash P[\vec{v}/\vec{w}] * H'$$

that is, the prover attempts to find a heap H' which satisfies this entailment. In terms of the method call, H' is the part of the current heap H which is not needed by the method execution, i.e., the *frame* of the call. The frame H' is then composed with the spec's postcondition Q to form the result of the method call. This mechanism is sound because we appeal to the frame rule of separation logic [13]. When doing frame inference formal parameters are substituted by the actual ones.

EXAMPLE 2. *Consider the class Cell of Section 2 and let us assume we are executing the call $x.set(7)$ in the following symbolic heap:*

$$Val(x, \{content{=}\ 3\}) * Val(y, \{content{=}\ 9\})$$

Here x and y are objects of class Cell. Recall the spec of the set method:

```
void set(int x):
{ | Val(this, {content= X̂}) }
{ | Val(this, {content= x}) };
```

[5] Class invocations are sometimes called static invocations in Java. We call them class invocations, so as not to confuse them with the statically determined calls to instance methods, such as **super** calls.

[6] Jimple also has `interfaceinvoke`; we treat this in the same way as `virtualinvoke`.

```
class LinkedList
{
  private NodeLL head;
  private NodeLL tail;

  void create()
  {
    head=null;
    while (true) {
      NodeLL n = new NodeLL();
      n.next=head;
      head=n;
    }
  }
  ....
}
```

Figure 7. Create method for LinkedList class.

The frame inference question for the theorem prover is to find a H' such that:

$$Val(x, \{content{=}\ 3\}) * Val(y, \{content{=}\ 9\}) \vdash Val(x, \{content{=}\ \hat{X}\}) * H'$$

The solution frame is $H' = \hat{X}{=}3 \wedge Val(y, \{content{=}\ 9\})$. Combining the post-condition of the spec with the computed frame we obtain

$$\hat{X} = 3 \wedge Val(x, \{content{=}\ 7\}) * Val(y, \{content{=}\ 9\})$$

as $\hat{X}$ is unused it is removed leaving precisely what we expect from this method call.

Rule Assignment 2 returns the resulting state given by Invoke where the special variable ret is replaced by x. Similarly, rule New exploits Invoke calling it with the constructor of class C.

Rearrangement. The symbolic execution rules manipulate object's fields. When these are hidden inside abstract predicates both Lookup and Mutation require the analyzer to expose the fields they are operating on. This is done by the function

$$\mathsf{rearr} : Heaps \times Var \times Sig \to \mathcal{P}(Heaps)$$

which exploits the frame inference of the theorem prover. It is defined as:

$$\mathsf{rearr}(H,\ x.\rho) = \{H' * x.\rho \mapsto \hat{v} \mid H \vdash H' * x.\rho \mapsto \hat{v}\}.$$

6. Fixed Point Computation and Abstraction

The jStar's symbolic execution module constructs the control flow graph of the input Jimple program. Then, for each node of the control flow graph it computes the set of *all* possible symbolic heaps in which the node can be during any execution of the program. These sets can be infinitely large.

In order to ensure termination of symbolic execution we apply abstraction in the spirit of abstract interpretation [8], and more specifically taking inspiration from Space Invader [9].

To explain the issue, consider the method create() in Figure 7. The second column of Table 3 depicts the heaps computed by symbolic execution at the while-loop point at different iterations and starting with the empty heap emp. The table shows that the number of $NodeLL$ predicates grows unboundedly, and therefore the fixed-point computation would diverge. However, following Space Invader [9] we can replace chains of concrete $NodeLL$ by the more abstract lseg predicate. We loose the information on the precise length of the lists which in many cases is unessential and we will ensure the convergence of the fixed point computation of the loop-invariant. This is done in the third column of Table 3. In the second iteration of the abstract execution we do not have any knowledge of the size of the list, and hence in the third iteration we have reached the fixed point. Abstraction is done by rewriting rules, also called *abstraction rules* which implement the function

$$\mathsf{abs} : Heaps \to Heaps$$

We apply abstraction after the execution of any command, this helps to keep the state space small.[7] Usually abstraction rules are of the form:

$$\frac{\text{condition}}{H * H' \rightsquigarrow H * H''} \text{ (Abs Rule)}$$

that is H' is replaced by H'' if the condition holds. H'' is more abstract than H' since some unnecessary information is removed (abstracted away). In general, the simplification of the formula is done by removing existentially quantified variables that do not appear anywhere else in the heap. A concrete example is the following abstraction rule:

$$\frac{\hat{x} \notin Var(H, x)}{H * NodeLL(x, \hat{x}, \hat{v}) * \mathsf{lseg}(\hat{x}, \mathsf{nil}, \hat{v'}) \rightsquigarrow H * \mathsf{lseg}(x, \mathsf{nil}, \hat{v''})} \text{ (NL)}$$

This rule abstracts away the information that we have at least two nodes and replaces it with the knowledge that there is at least one node.[8]

For soundness, the abstraction rules must be true implications in separation logic: the more concrete heap should imply the more abstract one (e.g., in the rule above the left-hand side heap implies the right-hand side one).

A framework for abstract interpretation of Java programs. We have designed jStar to be very general: that is, we do not have hard-wired abstraction in our symbolic execution (as for example in Space Invader [9]). Instead, we introduce a mechanism to define new abstraction rules which can be understood by the theorem prover as special kind of logical rules. In this way, new abstract domains, in the sense of abstract interpretation, can be easily defined by providing new sets of abstraction rules. This approach provides jStar with a great level of flexibility.

The abstraction rules accepted by the theorem prover rewrite a frame inference question into a simpler one. They have the form:

$$\frac{\text{condition}}{H \vdash \mathsf{emp} \rightsquigarrow H' \vdash \mathsf{emp}} \text{ (jStar Abs)}$$

where the optional condition enforces that some variables do not appear anywhere else in H as in the example rule (NL) above. However, here we have a different format since a formula is simplified not directly but as a consequence of the simplification of an entailment.

The mechanism of abstraction by means of the jStar's abstraction rules works as follows. Let's assume we want to abstract the heap $H * H''$. For this purpose, the theorem prover is asked to find the frame of the entailment

$$H * H'' \vdash \mathsf{emp}$$

Note that since the right-hand side is emp, the sought frame is trivially $H * H''$. However, suppose we have (jStar Abs) among the set of abstraction rules. H matches with the left hand side of

[7] Another possibility would be to apply abstraction at loop points only. However, experimental experience has shown that this results in slower analyses.

[8] Here we use non empty list segment predicates composed by at least one node.

$$\frac{}{H, \quad x = E \longrightarrow x = E[\hat{x}/x] \wedge H[\hat{x}/x]} \text{ Assignment 1}$$

$$\frac{}{H * x.\langle C\colon t\ f\rangle \mapsto E_1, \quad x.\langle C\colon t\ f\rangle = E_2 \longrightarrow H * x.\langle C\colon t\ f\rangle \mapsto E_2} \text{ Mutation}$$

$$\frac{}{H * E.\langle C\colon t\ f\rangle \mapsto E_1, \quad x = E.\langle C\colon t\ f\rangle \longrightarrow x = E_1[\hat{x}/x] \wedge (H * E.\langle C\colon t\ f\rangle \mapsto E_1)[\hat{x}/x]} \text{ Look-up}$$

$$\frac{}{H, \quad \mathsf{return}\ E \longrightarrow ret = E \wedge H} \text{ Return}$$

$$\frac{S \in \mathsf{spec}_{\mathsf{invoke}}(C, t, m) \qquad \mathsf{jsr}(S, H, v) = H'}{H, \quad \mathsf{invoke}\ x.\langle C\colon t\ m\rangle(v) \longrightarrow H'} \text{ Invoke}$$

$$\frac{H,\ \mathsf{invoke}\ y.\langle C\colon t\ m\rangle(v) \longrightarrow H'}{H, \quad x = \mathsf{invoke}\ y.\langle C\colon t\ m\rangle(v) \longrightarrow H'[x/ret]} \text{ Assignment 2}$$

$$\frac{H[\hat{x}/x],\ \mathtt{virtualinvoke}\ x.\langle C\colon void\ \mathsf{init}\rangle(v) \longrightarrow H'}{H, \quad x = \mathsf{new}\ C(v) \longrightarrow H'} \text{ Allocation}$$

Table 2. Symbolic execution rules for basic command. The primed variable $\hat{x}$ is fresh.

Iteration	Concrete execution	Abstract execution
1	$head = n \wedge NodeLL(n, \mathsf{nil}, \hat{v})$	$head = n \wedge NodeLL(n, \mathsf{nil}, \hat{v})$
2	$head = n \wedge NodeLL(n, \hat{n}, \hat{v}) * NodeLL(\hat{n}, \mathsf{nil}, \hat{v'})$	$head = n \wedge \mathsf{lseg}(n, \mathsf{nil}, \hat{w})$
3	$head = n \wedge NodeLL(n, \hat{n}, \hat{v}) * NodeLL(\hat{n}, \hat{n'}, \hat{v'}) * NodeLL(\hat{n'}, \mathsf{nil}, \hat{v''})$	$head = n \wedge \mathsf{lseg}(n, \mathsf{nil}, \hat{w})$
4	...	
5	...	

Table 3. Computed heaps at while-loop point of method create in concrete and abstract execution.

(jStar Abs), therefore, the rule fires and the entailment question is replaced by

$$H' * H'' \vdash \mathsf{emp}$$

where, presumably, H' is more abstract than H. As noted above, since the right-hand side is emp, the trivial frame for this entailment is $H' * H''$. The latter is then returned by the prover as an abstraction of $H * H''$.

An example of few useful abstraction rules we use to deal with lists and their values is reported in Table 5. Here they are specified with the syntax used in the abstraction file given as input to jStar. The initial frame inference question is indicated on the top (just below the name of the rule). The resulting entailment is at the bottom after the **if** keyword. The optional condition is specified by the "where" clause. The keyword "notincontext" following a variable informally means: that variable does not occur in any other predicate in the symbolic heap. The keyword "notin" prevents two variables to be unified. Hence the rule LS_LS should be read as: replace the entailment $\mathsf{lseg}(x, \hat{x}, s_1) * \mathsf{lseg}(\hat{x}, \mathsf{nil}, s_2) * H \vdash \mathsf{emp}$ by $\mathsf{lseg}(x, \mathsf{nil}, s_1 \cdot s_2) * H \vdash \mathsf{emp}$ provided $\hat{x} \notin Var(H) \cup \{x\}$ for any heap H. The other rules can be explained as follows.

LS_OBS If an observer ?w is the first elements of a list of unknown type starting at ?x, i.e. ls(?x,?z,cons(?w,?r)), then we can replace the occurrence of the single observer with a non-empty list of observers (i.e. lsObs) starting at ?w and having same subject and value, followed by a possibly empty list (lspe) of unknown type.

LS_OBS_APP1 This rule simply replaces two consecutive lists of observers with the same subject and value by one list which is the append of the two original ones.

Built-in heap normalizations. Some natural simplifications of symbolic heap have been built in the symbolic execution. First of all, Jimple $-variables are existentially quantified after their use. Moreover, the other abstractions are:

$$\frac{}{P(\vec{x}) \wedge P(\vec{x}) \wedge \Pi \wedge \Sigma \rightsquigarrow P(\vec{x}) \wedge \Pi \wedge \Sigma} \text{ Built-in 1}$$

$$\frac{}{E = \hat{x} \wedge \Pi \wedge \Sigma \rightsquigarrow (\Pi \wedge \Sigma)[E/\hat{x}]} \text{ Built-in 2}$$

Table 4. Basic built-in abstraction for the pure part.

1. Erasing multiple occurrences of same predicate in the pure part.
2. Abstracting unneeded primed variables from the pure part.

The latter two built-in abstractions are formalized by the rules in Table 4.

7. Related Work

The most closely related work is by Chin *et al.* [7]: they have also built a tool for verifying object-oriented programs with separation logic. They also distinguish between static and dynamic specifications. However, underlying their tool is the standard class invariant approach, and we believe this makes it difficult for them to express the specifications for the design patterns verified in this paper. We do not believe they can verify the examples we have presented.

Both Spec♯ [2] and JML [6, 17] have been used to specify and verify object-oriented programs. There have been several extensions (for example, [3, 18, 21]) to both systems proposed to handle the kind of examples we have presented in this paper. We do not believe that either system can currently handle all the examples presented in this paper.

Smans *et al.* [28] and Rosenberg *et al.* [1] have both proposed ways of automating the ideas of dynamic frames [14]. Dynamic

Rule LS_LS:
| | ls(?x,_x,?s1) ∗ ls(_x,nil(),?s2) ⊢ |
where
_x notincontext;
_x notin ?x
if
| | ls(?x,nil(),app(?s1,?s2)) ⊢ |

Rule LS_OBS:
| | ls(?x,?z,cons(?w,?r)) ∗ Observer(?w,{val=?v; subject=?s}) ⊢ |
if
| | lspe(_f,?z,?r) ∗ lsObs(?x,_f,cons(?w,empty()),?v,?s) ⊢ |

Rule LS_OBS_APP1:
| | lsObs(?x,_f,?l,?v,?s) ∗ lsObs(_f,nil(),?l2,?v,?s) ⊢ |
where
_f notincontext;
_f notin ?x, ?l, ?v, ?s, ?l2;
if
| | lsObs(?x,nil(),app(?l,?l2),?v,?s) ⊢ |

Table 5. A sample of abstraction rules dealing with lists and values.

frames bring many of the advantages of separation logic to first-order theorem proving, and may enable them to specify and verify examples like these.

Similar to how jStar allows new abstract domains to be defined by new abstraction rules, TVLA [19] is a parametric tool for defining shape analyses which uses first-order logic with transitive closure. The user, by defining so-called *instrumentation predicates*, changes the way the abstraction is done. TVLA is based on the concept of *canonical abstraction* [27], therefore, the fundamental abstraction principle is fixed. Depending on the problem to be analyzed, instrumentation predicates are used to prevent the loss of crucial information. jStar takes the opposite point of view: without abstraction rules nothing is abstracted. Abstraction rules are used to explicitly state which unnecessary information can be abstracted away. Most importantly, however, TVLA is a system oriented to define static analyses whereas jStar is oriented towards verification.

Bogor [10, 30] is an extensible explicit-state model checking framework for Java. Bogor is completely automatic, but only considers heap structures of bounded size. On the other hand, jStar, by using appropriate abstractions, can deal soundly with unbounded heaps. We believe that, in special cases, it would not be unrealistic thinking of using jStar in "Bogor style", i.e., by disallowing abstraction and executing only on concrete state spaces. ✯

8. Future Work

At the time of writing, jStar is just a research prototype and, therefore, there is plenty of room for improvement. One of the strengths of jStar is its flexibility, which gives us a substantial power for experimentation with new ideas and techniques. Using different sets of logic/abstraction rules we can obtain different ways of reasoning about programs or doing abstraction. However, this high-flexibility of our system might raise problems. Currently, users are required to have some knowledge of theorem proving in order, for example, not to design unsound logic rules (therefore, at the moment, jStar might be too complex to use by programmers).

In the future, we are planning to study several kinds of possible automation which may help in alleviating this problem.

- We are planning to design several sets of logic and abstraction rules, able to cover a wide range of programs, and possessing good properties (e.g., proven to be sound, ensuring termination and progress etc.).
- We are going to investigate the possibility of mechanizing some steps for producing new rules in a sound way.
- Further automation can be provided by studying techniques for inferring method specifications.

We believe that these features have the potential to reduce the danger of untrained users introducing errors. ✯

9. Conclusions

In this paper we described jStar, a new automatic verification tool for Java programs, and the theory behind it. jStar's foundations rely on the combination of new separation logic advances in theorem proving, symbolic execution, and abstraction.

jStar is almost completely automatic. It requires generally small straightforward pre/post annotations, and loop-invariants are computed automatically.

The practical results on real-world programs are very promising. Using jStar, we have been able to verify an entire implementation of four design patterns. Although used commonly when implementing Java applications, until now, these patterns have been beyond the reach of other state of the art automated Java verification approaches. ✯

Acknowledgments. We would like to thank Peter O'Hearn for strongly encouraging us in writing this paper. We also would like to thank the anonymous referees for many interesting suggestions which helped us to improve the paper.

The authors have both been supported by Royal Academy of Engineering Research Fellowships. ✯

References

[1] A. Banerjee, D. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In *Proceeding of ECOOP*, volume 5142 of *LNCS*, pages 387–411. Springer, 2008.

[2] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec♯ programming system: An overview. In *Proceedings of CASSIS*, pages 49–69, 2005.

[3] M. Barnett and D. A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *MPC*, volume 3125 of *LNCS*, pages 54–84. Springer, 2004.

[4] J. Berdine, C. Calcagno, and P. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO 2005*, volume 4111 of *LNCS*, pages 115–137. Springer, 2006.

[5] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In *Proceedings of APLAS*, volume 3780 of *LNCS*, pages 52–68. Springer, 2005.

[6] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. In *Proceedings of FMICS*, pages 73–89, 2003.

[7] W.-N. Chin, C. David, H. Nguyen, and S. Qin. Enhancing modular OO verification with separation logic. In *Proceedings of POPL*, pages 87–99. ACM, 2008.

[8] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77: Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[9] D.Distefano, P.O'Hearn, and H.Yang. A local shape analysis based on separation logic. In *Proceedings of TACAS*, volume 3920 of *LNCS*, pages 287–302. Springer, 2006.

[10] M. Dwyer, J.Hatcliff, M.Hoosier, and Robby. Building your own software model checker using the bogor extensible model checking framework. In *CAV*, volume 3576 of *LNCS*, pages 148–152. Springer, 2005.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[12] M. Grand. *Patterns in Java*. Wiley, second edition, 2002.

[13] S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, pages 14–26, 2001.

[14] I. T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *FM*, volume 4085 of *LNCS*, pages 268–283. Springer, 2006.

[15] N. R. Krishnaswami, J. Aldrich, and L. Birkedal. Modular verification of the subject-observer pattern via higher-order separation logic. In *Proceedings of FTfJP*, 2007.

[16] G. Leavens, K. Leino, and P.Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 19(2):159–189, 2007.

[17] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

[18] K. R. M. Leino and W. Schulte. Using history invariants to verify observers. In *Proceedings of ESOP*, volume 4421 of *LNCS*, pages 80–94. Springer, 2007.

[19] T. Lev-Ami and M. Sagiv. Tvla: A system for implementing static analyses. In *SAS*, volume 1824 of *LNCS*, pages 280–301. Springer, 2000.

[20] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM TOPLAS*, 16(6):1811–1841, 1994.

[21] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62:253–286, 2006.

[22] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of CSL*, volume 2142 of *LNCS*, pages 1–19. Springer, 2001.

[23] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL*, pages 268–280, 2004.

[24] M. J. Parkinson. *Local Reasoning for Java*. PhD thesis, Computer Laboratory, University of Cambridge, 2005. UCAM-CL-TR-654.

[25] M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In *POPL*, pages 247–258, 2005.

[26] M. J. Parkinson and G. M. Bierman. Separation logic, abstraction and inheritance. In *Proceedings of POPL*, pages 75–86. ACM, 2008.

[27] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM TOPLAS*, 20(1):1–50, 1998.

[28] J. Smans, B. Jacobs, F. Piessens, and W. Schulte. An automatic verifier for java-like programs based on dynamic frames. In *Proceedings of FASE*, volume 4961 of *LNCS*, pages 261–275. Springer, 2008.

[29] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[30] X.Deng, J.Lee, and Robby. Bogor/kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *ASE 2006*, pages 157–166. IEEE, 2006.