The VeriJava Programming System: An Overview

Jianjun Zhao, Cheng Zhang, Sibo Zhang, and Jiaming Zhang

Deparment of Computer Science and School of Software
Shanghai Jiao Tong University
1311 Software Building, 800 Dongchuan Road, Shanghai 200240, China
zhao-jj@cs.sjtu.edu.cn

Abstract. VeriJava is a novel programming system, which extends Java language, with just a few new language constructs, to support adding contracts to Java. VeriJava consists of an object-oriented programming language called VeriJava, a compiler for compiling VeriJava programs to Java bytecode, and a static verifier for assuring that the VeriJava code is consistent with its specifications. This paper discusses the goal and the overall programming approach of VeriJava. The paper also gives examples of VeriJava programs. On one hand, VeriJava provides a way with assertions (pre- and postconditions, and class invariants), to support runtime checking such as debugging and testing. On the other hand, VeriJava also offers the possibility of automatic compile-time verifications such as checking the code of a method against its specification, checking that the specification of a subclass is compatible with the specification of its superclass.

1 Introduction

The principle of Design by Contract (DBC) was originally introduced by Meyer [16] for supporting reliable software development. The current research so far in supporting the principle of DBC in Java is focused mainly on the development of specific specification languages [15, 7, 13] and tools [11, 2, 14]. However, little research has been focused on adding contracts to Java from a language design viewpoint. Until recently, only a few programming languages, e.g., Eiffel [16, 17] and Spec# [1] implement pre- and postconditions, class invariants in executable code so that they are checked at compiling time and/or run time. However, Java does not have such support, so programmers who want to use pre- and postconditions, and class invariants in Java often write comments documenting the conditions. This is not an ideal situation because the comments are not verified automatically, and they may not even be consistent with the actual code. This motivates us to design a new programming system that extends Java with contract specifications such as preconditions, postconditions, and class invariants so that programmers who write programs in Java can easily write contracts as well.

The language we designed is called VeriJava which is a programming system for adding contracts to Java [8]. The system consists of an object-oriented programming language called VeriJava, a compiler for compiling VeriJava programs to Java bytecode, and a static verifier for assuring that the VeriJava code is consistent with its specifications. VeriJava is a simple and practical extension to Java language, with just a few new language constructs, to add contracts to Java from a language design level. VeriJava also supports class specification inheritance.

This paper discusses the design rational and the overall programming approach of VeriJava. The paper also gives examples of VeriJava programs. On one hand, VeriJava provides a way to Java with assertions (pre- and postconditions, and class invariants), supporting runtime checking such as debugging and testing of VeriJava programs. On the other hand, VeriJava also offers the possibility of automatic compile-time verification of VeriJava programs such as checking the code of a method against its specification, checking that the specification of a subclass is compatible with that of its superclass. VeriJava is currently under development at Shanghai Jiao Tong University.

This paper makes the following contributions:

- **Language extension**: It presents a small, yet seamless extension to Java to support contract specifications and non-null types.
- **A novel programming system**: It designs and implements a novel verifiable programming system which include the VeriJava language, compiler, program verifier, and runtime checker, to support formal specification and verification of Java-like object-oriented programs.
- **Tool Support**: It constructs a toolkit as an Eclipse plug-in to support static and dynamic checking of VeriJava programs.

The rest of the paper is organized as follows. Section 2 discusses our design rationale for VeriJava. Section 3 briefly introduces the VeriJava language with some examples. Sections 4 and 5 describes some implementation issues of VeriJava system and its Eclipse plug-in toolkit respectively. Section 6 discusses some related work, and concluding remarks are given in Section 7.

2 Design Rationale

Our purpose to develop VeriJava is to study

- (1) how to add contracts to Java by integrating assertions in the Java language itself and
- (2) how to verify Java code at both compiling time and run time.

Because of this goal, designing the VeriJava language is really just part of our project. We must be able to develop techniques and tools to support formal verification of VeriJava programs. To make this possible, we have chosen to design VeriJava as a compatible extension to Java so that

- (1) it will facilitate adoption by current Java users, and
- (2) it will facilitate adoption of existing Java-based tools to check VeriJava programs.

When we design VeriJava, we keep the following issues in mind in order to make VeriJava compatible with Java.

- Each legal Java program must be a legal VeriJava program.
- Each legal VeriJava program must run on standard Java virtual machines.

- Programmers who write VeriJava programs must feel like a natural extension of Java programs.
- It must be possible to extend existing Java tools (such as IDEs, documentation tools, and design tools) to support VeriJava in a natural way.

3 The Language

The VeriJava language is a superset of Java. It is a simple and practical extension to Java with just a few new language constructs to implement contract specifications in Java. It combines the specification styles of JML [15] and Spec# [1] which use `requires` and `ensures` keywords to describe pre- and postconditions for methods and use `invariants` to describe invariants for classes. VeriJava also supports non-null types in Java.

The contracts in VeriJava should have no side-effect on the program, which means that the status of the program should not be modified by the contracts. Methods with contracts in VeriJava can be used to form the contracts of other methods. On the other hand, not all types of methods in VeriJava should have contracts. For example, private methods, which can not be accessed by external requesters, need not to have contracts. Since a private method can only be accessed by other methods within the same class it belongs to, whose consistency of contracts will be checked when the public or other type of methods are invoked.

3.1 Method Specifications

In VeriJava, the specification of a method is similar to that of a method in JML [15] which is composed of three formulas, that is, a precondition, a postcondition, and a frame condition that is declared by a `requires`, `ensures`, or `modifies` keyword respectively. The precondition, postcondition, and frame condition together form a *specification* of the method, that can be used to verify the code of the method.

Preconditions Preconditions specify conditions that must hold before a method can execute. As such, they are evaluated just before a method executes. Preconditions involve the system state and the arguments passed into the method. If the precondition is not held properly, it will be the requester's responsibility. VeriJava uses `requires` construct to specify a precondition for a method. As an example, the following shows a simple precondition for method `m`.

```
class A {
  int a = 1;
  void m(int x)
     requires x > 100 && x <= 200
  {
      a = x;
  }
}
```

The precondition (represented by a `requires` statement) specifies that the value of variable `x` must be greater than 100 and less or equal to 200 at the call of method `m`.

Postconditions Postconditions specify conditions that must hold after a method completes. Consequently, postconditions are executed after a method completes. Postconditions involve the old system state, the new system state, the method arguments, and the method's return value.

There are two different types of postconditions in VeriJava, that is, the *normal return condition* and the *exceptional return condition*. For a normal return, the contracts, which is represented with `ensures` statement, will be checked when the method has executed successfully without any exceptions thrown. For a exceptional return, the contracts, which is represented with `signals` statement, will be checked when the method is terminated by any exception.

VeriJava uses `ensures` construct to specify a postcondition for a method. As an example, the following shows a simple postcondition for method `setX`.

```
class A {
  int a;
  public void setX(int x)
      requires x > 10;
      ensures a == old(a) + x;
      modifies a;
  {... }
}
```

The postcondition (represented by an `ensures` statement) specifies that the value of variable `a` must equal to that of the sum of old `a` and variable `x` after the execution of the method `setX`.

Frame Conditions Frame conditions specify the fields that will be modified by a method, which means that the method will change the status of this object. Private fields inside a class can not be seen from the outside. But if the field is modified by some method that will be invoked by other classes, it must be explicitly specified. In addition, if a method modifies collections or arrays of a class, it implicitly means that this method can modify the elements of the collections or arrays. VeriJava uses `modifies` construct to specify a frame condition for a method.

3.2 Class Specifications

VeriJava also supports the class invariant specification for expressing global properties for the instance of a class as a whole. An invariant of a class is a set of assertions (i.e., `invariant` clauses) that the instance of this class will satisfy at all times when the state is observable.

Class Invariants Borrowed from JML [15] and Spec#[1], VeriJava uses an `invariant` keyword as well to specify a class invariant. As an example, the following shows a simple invariant for class `A`.

```
class A {
  int a;
  invariant a > 0;
  public void setVal(int val)
      requires val > 10;
      ensures a == old(a) + val;
  {
      a += val;
  }
}
```

This class invariant (represented by an `invariant` statement) states that the value of variable `a` must be greater than zero for all the objects of class `A`.

Note that the invariant condition should not always be held. When the method is executing and the object is at a status of unsteady, the invariant can be broken. However, the invariant conditions must be held before or after the executing of the method. In Spec#, there is an *expose block* which can be used to specify when an invariant can be broken. In VeriJava, however, we have not supported this issue currently due to some technical reasons.

Class Inheritance According to Java inheritance rules, we design some specification inheritance rules for VeriJava as follows.

- A subclass inherits the specifications of its superclass's public and protected members (fields and methods), as well as the public and protected class invariants.
- A subclass inherits the specifications of its superinterface's public methods, as well as interface invariants.

These inheritance rules ensure that a subclass specifies a behavioral subtype of its supertypes [6]. These inheritances can be thought of textually, by copying the public and protected specifications of the methods of a class's superclasses and all interfaces that a class implements into the class's specification and combining the specifications using `also` keyword, introduced in JML [15]. By the semantics of method combining using `also`, in addition to any explicitly specified behaviors, these behaviors must all be satisfied by the method.

3.3 Non-Null Types

Since large numbers of defects and errors in Java are caused by the null reference of an instance, VeriJava defines Non-Null type to make sure the non-nullity of every field in an object. Suppose `C` is a type which can be null, then the corresponding `C!` represents a non-null type. In VeriJava, this mechanism is implemented via the judgment of constructor, which requires the constructor explicitly initialize the fields of non-null types.

Not all kinds of fields can be declared as non-null types in VeriJava: this mechanism is invalid for static field and array. Because of the particularity of static fields' initialization in Java, the non-nullity of a filed can not be judged during compile time, which

should be checked at runtime. For an array, whose reference can be declared as not null, the initialization of the elements in it is often hard to present and meaningless. So the restriction of initialization in VeriJava does not contain the elements in an array.

4 System Implementation

The architecture of the VeriJava programming system consists of the compiler, the runtime library, and the static verifier. The system has been integrated into the Eclipse Environment as a plug-in in terms of directive, nature, editor, and compiler. Programmers can use VeriJava plug-in toolset to create VeriJava Engineering and VeriJava source. They can also use the editor to edit the VeriJava source file.

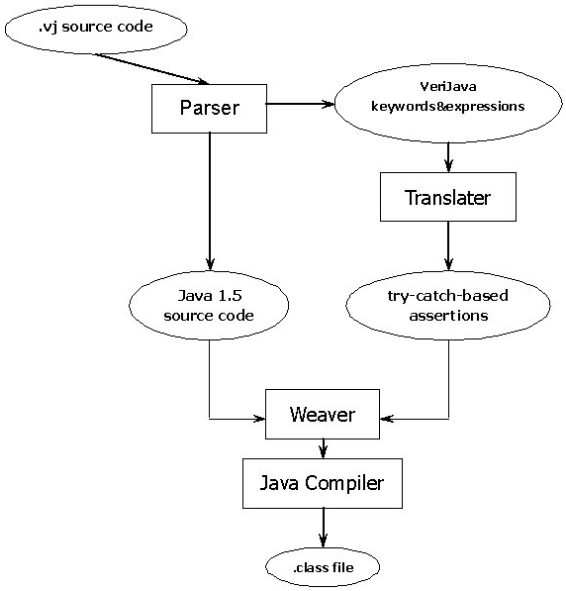

Fig. 1. The workflow of the compiler.

4.1 The Compiler

In order to express contracts, VeriJava extends Java programming language by introducing new keywords, syntax and semantics. It also extends the original Java compiler to process source code of the extended language. The compiler separates pure Java code and contracts in VeriJava source files and then processes them respectively. To support dynamic checking, the compiler inserts inline code into methods according to pre and post conditions and inserts synthetic methods into classes according to invariants. While

the program is running, the inline code is executed during certain method calls and the synthetic methods are called before and after the calls of certain methods, which can be implemented with extra code or interceptor mechanisms. In this way, contracts are checked during runtime.

The design of compiler can be divided into two parts. The first part is the translator which mainly take the responsible of weaving the try-catch-based assertions transformation of the contract into the binary code; the second part is the parser which resolve the VeriJava file and divide the source code into several parts. After the translator made the syntax analysis, the parser will reconstruct the whole source code with contract inside. The compiler also defined exceptions which will be thrown when runtime contracts checking does not pass.

Fig. 1 shows the main workflow in this compiler. As we can see, the parser takes the job of analyzing the structure of the VeriJava source code, and resolve it into the inner class named *VeriJavaClass*. The translator, which is the main part of the compiler, works only on the contract code and transforms it into try-catch-based Java code. After the transformation, the parser will reconstruct the code, so we can get a standard Java file output. This output stream will be sent into the JDK compiler so that we can eventually get an class file which can be run in any Java Virtual Machine.

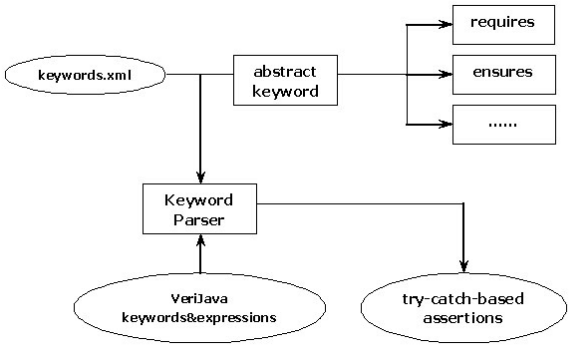

Fig. 2. The structure of the translator.

Fig. 2 shows the structure of the translator, which is constructed by an open architecture. As we can see, it include a configuration file `keywords.xml` which defines all the keywords with the corresponding class. So we can extend the keyword library in our further research of the syntax. When translator gets the contract code from the parser, it will throw the whole contracts into an circle which traverses all the keywords listed in the `keywords.xml` file. By using the feature of inheritance, we can transform every sentence with corresponding keyword into appropriate code. Let us see one simple example of the keyword `requires`:

```
class A {
  int a;
  public void setVal (int val)
     requires val > 10;
     ensures a == old(a) + val;
  {
       a+ = val;
  }
}
```

In this example, we can find that the method `setVal` has one parameter which `requires` greater than 10. The pre- and postconditions will be totally sent to the translator by main parser. After the transformation, we can get the code like this (not the real code):

```
class A {
   int a;

   public void setVal(int val) {
      try {
        the declaring of the VeriJava_Old function;
        if (val > 10) {
           a += val;
            if (a == VeriJava_Old(a) + val) {
              return;
           } else {
              throw new
              VeriJava.RuntimeException.KeywordException.EnsuresException("Ensures
                 not satisfied");
           }
       } else
           throw new VeriJava.RuntimeException.KeywordException.RequiresException("
                     Requires not satisfied");
       }
   } catch (Exception e) {
       System.out.println (e.printStackTrace());

   private int VeriJava_Old(int value) {
      ...
  }
}
```

As we can see that the code will be transformed into a try-catch-based pure java code with contract weaving inside. The function `VeriJava_Old` is configured which can resolve the old value inside the method, by declaring in front of the required code block.

The real code generation is much more complicated which includes also some parameter switch, inner library include, class type transform etc.

After the main transformation, the parser reconstructs the whole source code and throws the stream into the java standard compiler and generates the `.class` files.

The compiler also supports some parameters such as *-contract=on/off* which means we can compile a `.vj` file into a clean Java file without contracts when the verification is done without any exceptions. However, we suggest that the contracts should be included within the bytecode, as it will greatly improve the reliability of the code or even the whole system.

4.2 The Static Verifier

VeriJava compiler generates Java bytecode and encoded contracts as output and sends them to the static verifier that performs static checking. The procedure of static checking is quite similar to that of ESC/Java [7]. The static verifier transforms its input into verification conditions in the form of first order logic expressions, and then proves these expressions with a theorem prover and generates warning and error messages according to the result of proving. The static verifier consists of three major parts, including a translator, a verification condition generator and a theorem prover. Fig. 3 shows the structure of the static verifier.

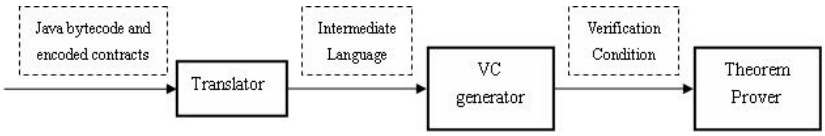

Fig. 3. The structure of the static verifier.

The translator first transforms VeriJava source code into an intermediate language based on Guarded Command language [4] like BoogiePL in Spec# [1]. The intermediate language is a simple language with only five types of statements, including assertion, assumption, assignment, sequential statements and optional statements as showed in Fig. 4. Precondtions in VeriJava are transformed into assumptions, postcondtions are transformed into assertions, assignments are preserved and other structures can be represented by sequential and optional statements. Although this intermediate language is quite simple, it can represents the semantics of Java quite well. For example, the following shows that a Java method called `getAge` can be translated into the code represented by the intermediate language:

```
public int getAge()
   requires  true
   ensures ( result == age)        assume true;
    && ( old(age) == age);  =>  ((assume age < 0; error:=1) | assume not(age<0));
{                               assert(age=age and _age=age);
   if (age<0) error=1;
   return age;
}
```

After generation of intermediate language code, the verification condition generator transforms it into verification condition which can be proved by a theorem prover. This transformation is based on calculation of weakest precondition. For a system, *S*, and one of its states, *Q*, the corresponding weakest precondition, *wp(S,Q)*, is defined as that if the initial state of *S* satisfies *wp(S,Q)*, *Q* is sure to be satisfied after execution of *S*. Accordingly, verification condition for a method, *M*, can be represented by weakest precondition, *wp(M, true)*. The rules of transformation is shown in Fig. 4.

By applying the rules given in Fig. 4 on the intermediate code, we can get the following verification condition for the method `getAge`.

S	*wp(S,Q)*
assert e	e∧Q
assume e	e→Q
x := e	Q(x := e)
A;B	wp(A, wp(B,Q))
A|B	wp(A,Q)∧wp(B,Q)

Fig. 4. Rules of Weakest Precondition Calculation

```
(age<0 ==> _age=age) && (not(age<0) ==> _age=age)
```

The static verifier takes advantage of an existing theorem prover to prove the verification condition. In our current version of VeriJava, we are using the Simplify theorem prover [3], which is also used by ESC/Java and Spec#, for our static verification of VeriJava code. We also consider using other theorem provers for our future versions of VeriJava.

4.3 The Run-time Checker

In VeriJava, after the compilation, contracts of each class are turned into contract checking codes inside each corresponding methods but not some independent methods or fields. This is because of some consideration of performance consumption. Some contract checking techniques for Java just turn contracts into some extra methods and fields which has been proved that performance is very poor. All such contract codes are rounded by try-catch block defined by us, so that it can be distinguished from the original code. When you execute the class with contracts, runtime checking will be performed. A specified exception will be thrown to indicate the detail information when some checking failure of contracts is encountered. Also the line of code that may contain errors will be indicated. An invariant is checked just before and after the execution of any public or protected method and should be held during the stable status, as we described previously.

5 Tool Support

We have constructed a toolkit for VeriJava based on Eclipse plug-in. Now many extension points have been implemented in this toolkit including wizards, project nature, package view, editor, action, and build. Programmers can easily use this toolkit to create VeriJava projects and source files. The naming principle of VeriJava source file is just the same as Java source files, but the extension of filename is `vj`. The editor can be used to edit VeriJava source code. Since the extension of build is implemented in this plug-in, whenever the file is saved, the source code will be compiled by the VeriJava compiler. Ordinary Java source file is considered as having all the contract conditions satisfied in this plug-in toolkit. Fig. 5 show the interface of the VeriJava editor on Eclipse plug-in.

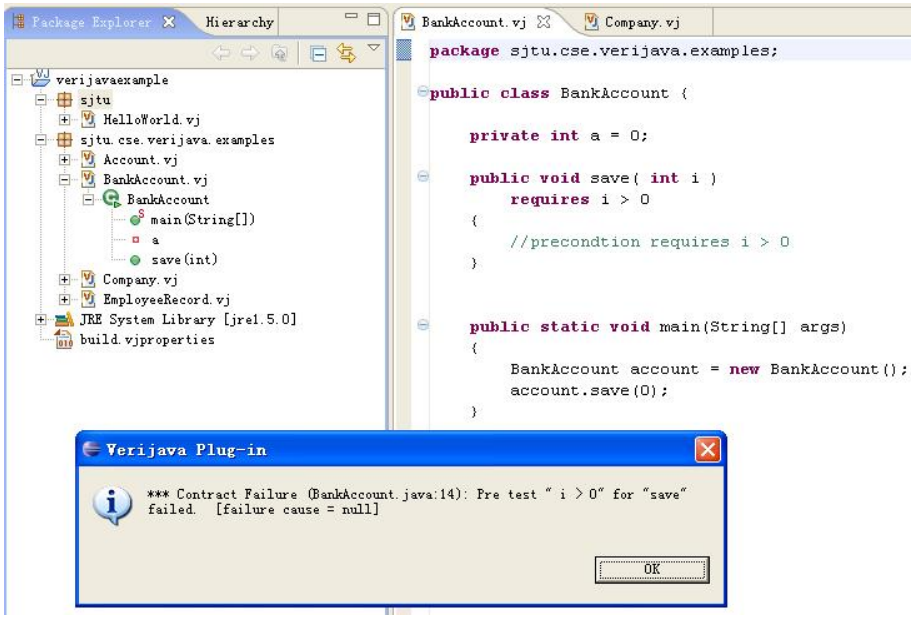

Fig. 5. The interface of the VeriJava editor.

We adopt an interpretation way of compiling in the design of compiler, which means that the contract conditions defined in VeriJava will be translated into some try-catch-based assertions after syntax analysis. Therefore, the output files generated by the compiler are still Java class files, whose file extension names are kept as classes. The contract conditions are weaved into the binary code which can be run on any Java Virtual Machine following J2SE specifications. Runtime checking can be executed based on the contracts in the classes.

6 Related Work

Several languages have been developed to support Design by Contract (DBC). Examples include Meyer's work on the programming language Eiffel [16, 17]. In Eiffel, one can specify pre- and postconditions for operations of abstract data types using Boolean expressions written in Eiffel. That is, unlike some interface specification language, program expressions are used in pre- and postconditions in Eiffel. In addition, in Eiffel, one can also specify the global properties of the objects of a class by using class invariants. Another example is Spec# [1], developed by Microsoft Research, which is a programming system for adding contracts to C# programming language. The Spec# consists of the object-oriented Spec# programming language which includes constructs for writing pre- and postconditions class invariants for methods and classes, the Spec# compiler which emits run-time checks to enforce these contracts, and the Boogie static program verifier which can check that a program is consistent with its specification.

Recently, the emergence of Java as a popular object-oriented programming language has eventually led to several projects designed for supporting DBC in Java. Examples include JML [15], ESC/Java [7], and AAL [13]. JML allows assertions to be specified for Java classes and interfaces, and provide very expressive power to specify Java modules (classes and interfaces). ESC/Java is a static checking tool for Java. It can statically check various errors in a Java program without executing the program. The annotation language in ESC/Java is a subset of JML that can be used to annotate Java code in various ways. AAL is an annotation language designed for annotating and checking Java programs. AAL provides the same opportunity as JML to support run-time assertion checking, and on the other hand AAL also supports full compiler-time checking for Java programs similar to ESC/Java. AAL translates annotated Java programs into Alloy [10], a simple first-order logic with relational operators, and uses Alloy's SAT solver-based automatic analysis technique to check Java programs. LOOP [11] is a project dedicated to verify JavaCard programs. LOOP adopts JML as its specification language for annotating Java modules, and transforms annotated Java programs into a theorem-prover, PVS [18], to statically check JavaCard programs. In addition to the projects mentioned above, several projects have also been carried out to support the DBC in Java including Jass [2, 12] and iContract [14]

Although these projects mentioned above can support DBC in Java, none of these uses the way by extending Java with contracts from a language design viewpoint.

Lackner *et al.* Lackner02 present an approach to supporting DBC in Java. Their approach is similar to ours in the sense that it also supports to write contracts such as pre- and postconditions for methods and class invariants for classes in an extended Java language. We see our work is different from Lackner *et al.*'s in several ways. First, in addition to support DBC in Java by extending Java language with some new language constructs, VeriJava also provides a static verifier and a run-time checker to check if a VeriJava program is consistent with its specification. Second, VeriJava supports non-null types in Java to make sure the non-nullity of every field in an object. Third, VeriJava is fully integrated into the Eclipse IDE as a plug-in toolkit.

7 Concluding Remarks

This paper presented our design of VeriJava, a novel programming system for adding contracts to Java. VeriJava is the combination of the design by contract approach of Eiffel [16, 17] and Spec# [1] and the interface specification language approach of the JML for Java [15]. We hope that by examining the similar ideas, as explored in JML and Spec#, of embedding contract specifications into Java to support verification, we can have a better understanding of how to support contracts in Java from a language design viewpoint and how our work can contribute to the grand challenge on The Verifying Compiler [9].

As our VeriJava project is just starting, there are still many problems that need to be solved in order to make our VieriJava system practical and useful. The future work for VeriJava project may include:

- to support contract specifications for concurrency.

- to refine our VeriJava language and develop a formal semantics for VeriJava to support static analysis, prototyping, and testing.
- to perform some experimental case studies on using VeriJava for programming.

References

1. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# Programming System: An Overview. *proc. CASSIS 2004*, LNCS vol. 3362, Springer, 2004.
2. D. Bertetzko, C. Fischer, M. Moller, and H. Wehrheim, "Jass - Java with Assertions," In K. Havelund and G. Rosu, editors, *ENTCS*, Vol. 55, Elsevier Publishing, 2001.
3. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A Theorem Prover for Program Checking. Technical Report HPL-2003-148, HP Labs, July 2003.
4. E. W. Dijkstra. A Discipline of Programming. Prentice Hall, Englewood Cliffs, NJ, 1976.
5. Martin Lackner, Andreas Krall, and Franz Puntigam. Supporting Design by Contract in Java. *Journal of Object Technology*, Vol.1, No.3, July 2002.
6. Barbaba Liskov and Jeannette Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Lnaugages and Systems*, Vol.16, No.6, pp.1811-1841, November 1994.
7. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pp.234-245, June 2002.
8. J. Gosling, B. Joy, and G. Steele. The Java Language Specification. The Java Series, Addison-Wesley, Reading, MA, 1996.
9. C. A. R. Hoare. The Verifying Compiler. *Journal of ACM*, Vol.50, No.1, pp.63-69, 2003.
10. D. Jackson. Alloy: A Lightweight Object Modeling Notation. *ACM Transaction on Software Engineering and Methodology*, Vol.11, No.2, pp.256-290, April 2002.
11. B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning About Java Classes (Preliminary Report). *Proceedings of ACM SIGPLAN 1998 Conference on Object-Oriented Programming Systems, Languages and Applications*, pp.329-340, October 1998.
12. The JASS project: `http://semantik.informatik.uni-oldenburg.de/~jass/`.
13. S. Khurshid, D. Marinov, and D. Jackson. An Analyzable Annotation Language. *Proceedings of ACM SIGPLAN 2002 Conference on Object-Oriented Programming Systems, Languages and Applications*, October 2002.
14. R. Kramer, "iContract – the Java Design by Contract Tool," *Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS-USA)*, 1998.
15. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary Design of JML: a Behavioral Interface Specification Language for Java. Technical Report TR98-06, Department of Computer Science, Iowa State University, 1998 (Last version: June 2002).
16. B. Meyer. Eiffel: The Language. Object-Oriented Series, Prentice Hall, New York, N.Y., 1992.
17. B. Meyer. Object-Oriented Software Construction. Prentice Hall, New York, N.Y., Second Edition, 1997.
18. S. Owre, J. M. Rushby, N. Shankar, and F. von Henke. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering*, Vol.21, No.2, pp.107-125, February 1995.