

Next Generation of Virtual Platforms

Suad Alagić and Mark Royer
Department of Computer Science
University of Southern Maine
Portland, ME 04104-9300
e-mail: {alagic, mroyer}@cs.usm.maine.edu

ABSTRACT

This paper is addressing three major and quite controversial issues critical for the next generation of virtual platforms for object-oriented languages. The first one is providing explicit support for parametric types in the virtual machine. Lack of proper support in the Java Virtual Machine is precisely the reason for violations in Java 5.0 with respect to dynamic type checking, correctness of run-time type information, and correctness of persistence and reflective capabilities. The second limitation of current object oriented languages and their virtual platforms is proper support for assertions (constraints). Contrary to customary views on assertions, we show that object-oriented languages (of the next generation) equipped with assertions and running on virtual platforms require proper support for these high-level features in the virtual machine. Parametric types equipped with assertions (constraints) should play a significant role in applications such as database systems, but the current virtual platforms have very inadequate support for persistence. We argue for much more sophisticated orthogonal persistent capabilities, where orthogonality is also a point of controversy. The main components of the required architecture are presented in the paper. This architecture includes representation of parametric types and logic-based constraints in class file and class object structures, a loader that performs appropriate actions on type parameters and constraints, and reflective capabilities that report information on type signatures and their associated constraints. A particularly surprising component of this architecture, not considered as a part of a virtual platform, is an interface with a program verification system that performs checking of constraints and verifies behavioral subtyping requirements. This is accomplished by interfacing the program verification system with reflective capabilities and performing verification by accessing type signatures and their associated constraints in (dynamically) loaded class objects. The specifics are elaborated in the paper and implemented by extending the Java Virtual Machine with the proposed functionalities.

1. EXTENDED VIRTUAL PLATFORMS

The core idea of this paper is that virtual platforms, for object-oriented languages, with parametric types and logic-based constraints (assertions) must provide proper support for these high-level language features.

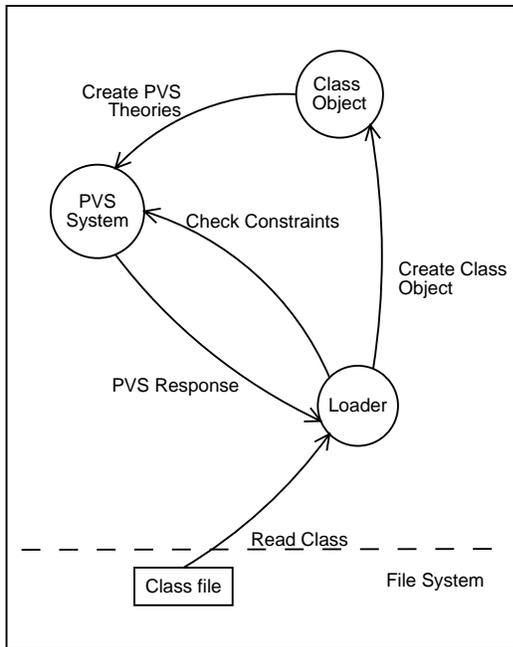
Lack of proper support for parametric types has major implications for correctness of static and dynamic type checking, correctness of run-time type information, and correctness of persistence and reflective capabilities of virtual platforms [18, 36]. This point of view comes as a surprise after years of research and experience with languages such as C++ which indicated that parametric polymorphism can and should be handled completely statically. Hence, no support in the virtual machine should be required. But we show that lack of proper support in the Java Virtual Machine is precisely the reason for all of the above anomalies in Java 5.0. These surprisingly severe implications have been properly understood by the designers of the C# extension with parametric polymorphism [25]. Unlike the JVM, Common Language Runtime [37] has been extended with a variety of features ensuring proper support for parametric polymorphism.

Almost complete lack of assertions in mainstream object-oriented languages such as Java (Eiffel [31] is an exception) is also regarded as a major limitation. But providing support for assertions comes with considerable more subtleties. Assertions (constraints) are logic-based, declarative, and hence a high-level language feature that seemingly does not require support in the underlying virtual machine. But just like with parametric types, we show that for languages running on virtual platforms this perfectly natural point of view does not hold any more.

The view that virtual platforms for object-oriented languages should provide proper support for the management of constraints is new and much more subtle. We show that lack of proper support for constraints has non-trivial implications on integrating constraints into the overall environment and managing them in their declarative form. Lack of proper support for constraints makes it impossible for reflective capabilities of a virtual platform to report information on constraints along with types, and on the ability to verify behavioral subtyping requirements. A major subtlety in languages on virtual platforms is that the source code may not be available. Hence, verifying behavioral subtyping requires access to a suitable representation of constraints in class files and

class objects. Because of dynamic loading these conditions must often be verified dynamically where static verification becomes a particular case. Of course, loading all class objects and performing effectively static verification is a preferable mode of operation in the proposed extended platform as well, especially if an interactive program verification system is used.

Parametric classes along with constraints are typically required in typed persistent and database systems because of various types of collections equipped with data integrity constraints. But the current most popular virtual platforms for object-oriented languages, the Java Virtual Machine in particular, have very inadequate support for persistence. This has led to a variety of specialized virtual machines that have those capabilities. We argue that much more sophisticated persistent capabilities are required in the next generation of virtual platforms. Whether those will in fact support orthogonal persistence is yet another point of controversy.



The main components of an extended virtual platform represented in the diagram above are:

- Class files that allow representation of parametric types along with their associated logic-based constraints.
- Class objects that contain correct actual type signatures along with constraints.
- A loader that assembles class objects from class files and properly manages type parameters and constraints.
- Reflective capabilities that allow introspection of parametric types and constraints.
- An interface with a program verification system which allows load-time checking of constraints and behavioral subtyping requirements.

In this extended platform the core agent is a sophisticated loader. The loader performs access to the parametric class

file equipped with assertions, creates a class object for a particular instantiation of a parametric class with specific actual type parameters, loads the constraints into the created class object, and then invokes the program verification system in order to verify behavioral subtyping requirements. The program verification system performs access to the loaded class objects through an interface which makes use of extended reflective capabilities. The interface component produces a program verification theory of a class, and the program verification system carries out deduction and reports the results to the extended loader.

One implication of the above architecture is that when class objects are promoted to persistence, unlike the situation in Java 5.0, those objects will contain correct type information. In addition, persistent class objects will be equipped with constraints.

2. GENERICITY IN JAVA CONTROVERSY

Lack of proper support in the virtual platform for parametric polymorphism will be illustrated by a major controversy in the most recent (5.0) release of the dominating object-oriented language Java. The accepted solution allows violations of the Java type system so that neither static nor dynamic type checking work correctly. Run-time type information is incorrect which makes dynamic type violations possible when using reflection. Persistence mechanisms do not work correctly either for the above reasons.

In the example below, a linked list of elements of type `Object` is assigned to a linked list of elements of type `Employee` in complete violation of the typing rule for the assignment statement.

```

public class UpdateTransaction{
    public static void main(String[] args) {
        LinkedList<Employee> employees =
            new LinkedList<Employee>();
        LinkedList objects = new LinkedList();
        objects.add(new Object());
        employees = objects; }
}

```

Contrary to the claims about Java 5.0, this example shows nontrivial problems when using legacy classes such as `LinkedList` and newly available parametric classes. If later on in the program we try a naive and correct statement `Employee emp = employees.remove();` this would fail unexpectedly with a run-time exception.

A simple program below promotes a collection of employees to persistence.

```

public class StoreCollectionObject {
    public static void main(String[] args)
        throws Exception {
        FileOutputStream fileout =
            new FileOutputStream("employees");
        ObjectOutputStream out =
            new ObjectOutputStream(fileout);
        Collection<Employee> employees =
            new LinkedList<Employee>();

```

```

    employees.add(new Employee());
    out.writeObject(employees); }
}

```

In a separate transaction, given below, an attempt to access this persistent collection is made in such a way that its type is misinterpreted on purpose. The compiler issues a warning in this type cast no matter what the instantiated parametric type in the type cast is, even if it is the correct one (`Collection<Employee>`)`in.readObject()`. This is wrong and contrary to the Java type system. Any type should pass a static check in the type cast because any type is a subtype of `Object`, which is the result type of `readObject`. Since the dynamic type information is incorrect, the incorrect type cast will succeed, and the class cast exception will not be thrown.

```

public class ReadCollectionObject {
    public static void main(String[] args)
        throws Exception {
        FileInputStream filein =
            new FileInputStream("employees");
        ObjectInputStream in =
            new ObjectInputStream(filein);
        Collection<Department> departments = null;
        try {departments =
            (Collection<Department>) in.readObject();
        } catch (ClassCastException e) { /* ... */ } }
}

```

Although the above problems are demonstrated for Java serializable objects, they will appear in other well-known models of persistence [10, 24, 16, 17, 23].

In the example below a transaction interrogates the types dynamically using Java Core Reflection and obtains wrong type information. Based on this incorrect information a transaction performs a wrong action. Java Core Reflection is not capable of detecting the type violation, and hence, it does not throw the illegal argument exception. The transaction thus fails at a completely unexpected place where no exception handling is provided.

```

public class ReflectiveTransaction {
    public void updateEmployees(
        LinkedList<Employee> collection) {
        Employee e = collection.remove();
        // unexpected ClassCastException!
    }
    public static void main(String[] args) {
        ReflectiveTransaction trans =
            new ReflectiveTransaction();
        Collection<Department> departments =
            new LinkedList<Department>();
        departments.add(new Department());
        try {Method method = trans.getClass().getMethod
            ("updateEmployees", departments.getClass());
            method.invoke(trans, departments);
        } catch (Exception e) { /* ... */ } }
}

```

Failure at completely unexpected places where exception handling is not provided rather than where a violation occurs and exception handling code is provided is one of the worst problems in Java 5.0. Compiler warnings are either indecisive, occur where there is no violation, or do not occur

at all where a violation actually happens. Standard runtime exceptions are not thrown where they should be, for example in type casts and when using Java Core Reflection.

One of the reasons for the above problems is an implementation idiom called type erasure. But the idiom itself is a consequence of the real problem: lack of the required functionalities in the virtual platform for management of parametric types. Java class files (defined in the JVM Specification and hence a component of the virtual machine) allow representation of Java classes, but parametric classes are actually not classes at all, and representation of type parameters, their bound types, and the actual type parameters for instantiated parametric classes have no support in the JVM. In the JVM, run-time type information is contained in class objects. Correctness of dynamic type checking (for type casts in particular) and correct functioning of Java Core Reflection depend upon correct run-time type information. But JVM class objects in Java 5.0 do not contain correct type information for instantiated parametric classes. In order to load correct information, an extended JVM loader is required. Finally, in order for Java Core Reflection to report information on parametric types, this component of the platform must also be extended. This also shows why the type of persistent class objects for instantiated parametric classes is incorrect. It is a consequence of incorrect run-time type information in class objects.

3. CONSTRAINTS

A sample parametric interface `Collection`, in the Java notation, extended with assertions is given below as an illustration. The mutator methods `insert` and `delete` that change the underlying object state are equipped with method preconditions and method postconditions in the Eiffel-like style. The postconditions for the methods `insert` and `delete` are in fact specified in a temporal logic style because they refer to both the current and the previous object states. The latter are denoted using the keyword `OLD` as in Eiffel. But the constraints are much more general than in Eiffel as they include features from first order predicate calculus, quantifiers in the first place. A distinctive feature of the extended virtual platform is representation and management of constraints in their logic-based, declarative form, rather than in the procedural form, as in Eiffel.

```

interface Collection<T>{
    public int count(T o);
    public boolean belongs(T o);

    public void insert(T o)
        ensures this.count(o) == OLD this.count(o) + 1 AND
        FORALL (T o1)
            NOT o1.equals(o) IMPLIES
                this.count(o1) == OLD this.count(o1);
    public void delete(T o)
        requires this.belongs(o)
        ensures this.count(o) == OLD this.count(o) - 1 AND
        FORALL (T o1)
            NOT o1.equals(o) IMPLIES
                this.count(o1) == OLD this.count(o1);

    invariant
        countAx: FORALL (T o)
            this.count(o) >= 0
}

```

```

belongAx: FORALL (T o)
    this.belongs(o) IFF this.count(o) > 0
}

```

As an illustration, a parametric class `Bag` equipped with assertions is constructed in such a way that `Bag` is a behavioral subtype of `Collection`.

```

class Bag<T> implements Collection<T> {
    /* private representation */

public void insert(T o)
    requires super.requires
    ensures super.ensures;

public void delete(T o)
    requires super.requires
    ensures super.ensures;

public Bag<T> union(Bag<T> B);
public Bag<T> intersect(Bag<T> B);

invariant
unionAx: FORALL (Bag<T> B, T o)
    IF this.count(o) >= B.count(o) THEN
        this.union(B).count(o) == this.count(o) ELSE
        this.union(B).count(o) == B.count(o)

intersectAx: FORALL (Bag<T> B, T o)
    IF this.count(o) >= B.count(o) THEN
        this.intersect(B).count(o) == B.count(o) ELSE
        this.intersect(B).count(o) == this.count(o)
}

```

Following the rules of behavioral subtyping, the preconditions of the mutator methods `insert` and `delete` remain the same as in the supertype `Collection`. The postconditions of these methods could be strengthened with respect to the postconditions in the supertype, but we have a particular case in the above example in which they remain the same. The invariant of the class `Bag` strengthens the invariant inherited from the supertype `Collection` by introducing additional axioms.

4. VERIFICATION TECHNOLOGY

A particular verification system that could be interfaced with the extended virtual platform is PVS (Prototype Verification System) [33]. The choice of PVS is based on the fact that PVS has a sophisticated type system with particular forms of subtyping and parametric polymorphism, and in addition it provides support for a variety of logics. The first order predicate calculus-based constraints appear in the code sample given below.

In order to verify properties of a class equipped with assertions, the class must be transformed into a specification that can be handled by PVS. PVS specifications are theories. A theory of a class consists of the type signatures of its methods represented in the standard functional style, along with a collection of logic-based constraints, which are sentences expressed in the chosen logic.

PVS is used in the extended platform to verify behavioral subtyping requirements. When a class is loaded, PVS checks

whether the class is a behavioral subtype of its superclass. This is a condition for semantic (behavioral) correctness of software reuse. If a class is a behavioral subtype of its superclass then objects of the (sub)class will behave as objects of the superclass when the substitution of the former by the latter is carried out. In the extended platform, type signatures and their associated constraints are accessed in class objects, and the verification task is performed either at compile time or upon loading at the latest.

The paradigm underlying PVS is functional, but mutator methods, i.e., methods that change the underlying object state are not functions. A mutator method is represented as a binary predicate that defines pairs of object states such that the second state may be obtained from the first by invoking the mutator.

A simple PVS theory `Object` is given below. The sentences of this theory consist of three standard axioms which state that equality, as specified by the method `equal`, is in fact an equivalence relation. Because these constraints are specified as axioms, any valid implementation will have these properties.

```

Object : THEORY
BEGIN Object: TYPE+
    equal: [Object, Object -> bool]
    reflexive: AXIOM (FORALL (x: Object): equal(x,x))
    symmetric: AXIOM (FORALL (x,y: Object): equal(x,y)
        IMPLIES equal(y,x))
    transitive: AXIOM (FORALL (x,y,z: Object):
        equal(x,y) AND equal(y,z)
        IMPLIES equal(x,z))
END Object

```

5. REFLECTIVE CAPABILITIES

The existing Java reflective capabilities allow introspection of type signatures. An extended virtual platform allows introspection of type parameters of parametric classes (and interfaces), their names, bound types, and actual types. More sophisticated extensions allow introspection of constraints. Constraints are reported by the extended reflective capabilities in their logic-based declarative style. This is a major extension of the existing virtual platforms. Reflective capabilities will report type signatures of parametric classes along with the semantic information expressed by constraints.

The structure of constraints is determined by their underlying logic basis. This is why the class `Sentence`, defined below, is abstract and its methods only report universally and existentially quantified variables. A sentence is a logical formula with no free variables, i.e., all its variables are quantified. Methods of the class `Variable` report names and types of variables.

```

public abstract class Sentence {
    public Variable[] getVariables();
    public Variable[] getExistentialVariables();
    public Variable[] getUniversalVariables();
    public abstract boolean
        evaluate(Object[] variables);
}

```

The basic building blocks for constructing expressions are terms. The class `Term`, specified below, is abstract to allow for a variety of possible forms of terms. A term has a type and a collection of free variables. Given values of these variables a term may be evaluated. However, the method `evaluate` in the class `Term` is abstract since the specific evaluation rule depends upon the form of a term.

```
public abstract class Term {
    public Class    getType();
    public Variable[] getVariables();
    public abstract Object
        evaluate(Object[] variables);
}
```

A message term consists of the receiver term, a method, and an array of arguments which are also terms. The specific evaluation rule amounts to substitution of arguments and invocation of the underlying method.

```
public class MessageTerm extends Term {
    public Term    getReceiverTerm();
    public Method  getMethod();
    public Term[]  getArguments();
    public Object  evaluate(Object[] variables);
}
```

Formulas are constructed recursively starting with atoms and applying the rules of a particular logic.

Extensions of the class `Class` allow access to type parameters as well as to the declared and inherited invariant. In fact, the JVM for Java 5.0 already contains the extensions that allow access to formal type parameters and their bounds in a somewhat different way.

```
public final class Class { ...
    public boolean  isParametric();
    public String[] getFormalTypeParameters();
    public Class[]  getActualTypeParameters();
    public Class[]  getBoundTypes();
    public Sentence getInvariant();
    public Sentence getDeclaredInvariant();
}
```

The extensions of the class `Method` allow access to (declared and inherited) preconditions, postconditions, and transition constraints. Note that the standard method `toString` will return a correct, actual type signature of a method.

```
public final class Method {...
    public Formula  getPostCondition();
    public Formula  getDeclaredPostCondition();
    public Formula  getPreCondition();
    public Formula  getDeclaredPreCondition();
    public TransitionConstraint
        getTransitionConstraint();
}
```

6. PERSISTENCE

A suitable model of persistence for Java would feature transparency, orthogonality, and reachability [11, 9]. Transparency means that low-level details of the persistence implementation architecture are hidden from the users. Orthogonality means that objects of any type may be persistent. Reachability provides a transparent guarantee that when an object is made persistent, all of its components, direct and indirect, are also promoted to persistence.

The model of orthogonal persistence proposed in this paper is based on associating persistence capabilities with the root class `Object` [3, 7] as illustrated below. This model is quite different from the approach for implementing orthogonal persistence adopted in ODMG [17], PJama [10, 24], GemstoneJ [16] or JDO [23].

```
public class Object {
    ...
    public final Object makePersistent();
}
```

As all other classes extend `Object`, either directly or indirectly, orthogonality is guaranteed. The model of *persistence in the root* is truly object-oriented because it is based on message passing and inheritance. However, it requires re-implementation of the class `Object` and recompilation of the whole Java platform [5].

A rough equivalent of the model of associating persistence capabilities with the root class `Object` in the current Java technology would be obtained by making the root class `Object` implement the interface `Serializable`. This, of course, is currently not the case. This is precisely the reason why the Java model of persistence is not orthogonal. In addition, the Java model of persistence is not transparent, because it requires a user to deal with files (`InputStream` and `OutputStream`), reading and writing objects, etc. Similar remarks apply to the model of persistence in Eiffel [30], which in fact is not part of the language. All the low-level details of the persistence implementation architecture should be transparent to the users, which is not the case in either language.

Consider the implications of availability of orthogonal persistence and logic-based constraints in a major application area: database systems. The database class given below offers features similar to the ODMG class `Database` and the PJama interface `PJStore` with a fundamental difference: *constraints*. The method `bind` of this class binds a name (the second argument) to an object of any type (the first argument of type `Object`). As `Class` extends `Object`, a database contains bindings for both classes and objects. The method `lookup` returns an object of a database bound to a name. Note that the precondition and the postcondition of the method `bind` specify the semantic relationship with the method `lookup`.

```
public class Database {
    public Database(String name);
    public final boolean isOpen();
}
```

```

public final void open()
    ensures this.isOpen();
public final void close()
    requires this.isOpen();
public final Object lookup(String name)
    requires this.isOpen();
public final boolean bind(Object x, String name)
    requires this.isOpen() AND
        this.lookup(name) == null
    ensures this.lookup(name).equals(x);
invariant
loopUpAx: FORALL (String n, Class c, Class[] classes)
    (this.getClasses().equals(classes) AND
     this.lookup(n).equals(c) IMPLIES
      classes.contains(c))
equalsAx: FORALL (String n, Class c) (EXISTS (int i)
    (c.equals(this.getClass().getClasses()[i]))) AND
    c.getName().equals(n) IMPLIES
    this.lookup(n).equals(c)
}

```

The invariant of the `Database` class given above guarantees that dynamic lookup and Java Core Reflection [21] report consistent information on both classes and objects. The semantics of the Java Core Reflection method `getClasses` is extended so that it reports all classes, including those that are dynamically added using the method `bind`. The same condition applies to fields. Additional invariants guarantee that dynamic lookup of a database reports consistent information for statically declared classes and objects of its schema. This condition is specified above just for classes, the conditions on objects (fields of a database class) is the same.

Orthogonal persistence allows objects of any type to persist. The problem is that a parametric class is not a Java type. Only classes obtained from a parametric class by instantiation with the actual type parameters are Java types. In the architecture proposed in this paper parametric classes appear only in class files. When objects of instantiated parametric classes are promoted to persistence, their class objects are promoted to the persistent store along with objects by reachability.

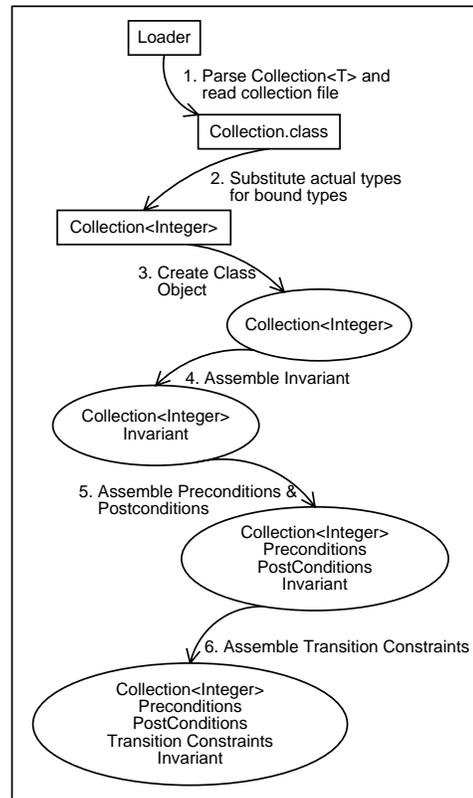
7. LOADER ACTIONS

The core component of the extended virtual platform is a sophisticated loader. A distinctive feature of the extended loader in comparison with the existing JVM native loader is that the extended loader performs correct management of type information when loading instantiated parametric classes. A more sophisticated extension is management of constraints. This includes loading constraints into the constructed class objects and invoking the program verification system to verify behavioral subtyping requirements.

The actions of the extended loader represented in the diagram below are:

- Parse the class name to separate the name of a parametric class from the actual type parameters.
- Find the parametric class file and create a class object for the instantiated parametric class.

- Load the type parameters block from the optional parametric class attribute with the formal, actual, and bound types for each type parameter.
- For each method and field of this `Class` object, load the actual type signatures obtained by substituting the formal type parameters with the actual type parameters.
- Assemble the code, local variable table, and the array of exceptions from the relevant predefined attributes of the class file.
- Assemble the invariant from the optional attribute `Formula` in the class attributes of the class file.
- For each method of this class, assemble the precondition and the postcondition from the optional attribute `Formula` in the attributes of the method.
- For each method of this class, assemble the transition constraint from the optional attribute `Formula` in the attributes of the method.



When a class object is loaded, introspection of type signatures and constraints in the class object is performed, and a PVS theory is constructed in order to perform verification. PVS signatures are specified in the standard algebraic (functional) form which is produced from the object-oriented form accessed by the extended reflective capabilities. The inheritance relationships are not available in PVS, hence, they must be represented by importing the theory of the superclass and making it a PVS supertype of the type represented by the newly constructed class object. Constraints must be transformed into the standard functional form. Terms are obtained from their object-oriented form reported by the extended reflective capabilities.

8. RELATED RESEARCH

The view that parametric polymorphism requires proper support in the underlying virtual platform has emerged slowly up to a full-fledged industrial implementation [25]. This view is somewhat a surprise given the general belief based on the initial research on parametric polymorphism and C++ implementation where parametric polymorphism is handled completely statically. But this widely accepted view does not hold for languages running on virtual platforms. Early proposals for extending Java with parametric polymorphism made different extensions to the JVM and did not survive [1, 32]. Subsequent research showed the implications of solutions to this problem that do not perform the required extensions of the JVM [36, 18].

Two current industrial solutions for extending an object-oriented language with parametric polymorphism are sharply different. In the C# solution, non-trivial changes are made to the underlying virtual machine (Common Language Runtime [37]) in order to accommodate parametric polymorphism [25]. In Java 5.0 only very limited and inadequate changes are made.

In the C# solution the language of the virtual machine (CLR) has been changed by adding new types to the intermediate language type system. Parametric forms of intermediate language declarations are introduced along with ways of referencing fields, classes, interfaces, and methods, and by specifying new instructions as well as generalizing the existing ones to the parametric forms. Our solution has been developed under much more restrictive assumptions: the instruction set, the run-time type introspection, dynamic dispatch, and class file structure all remain intact. The extensions in Java Core Reflection, internal class object representation, and the loading system have no impact on the correct functioning of Java legacy systems. The most important difference is that our technique extends to representation and management of assertions with all the above explained advantages with respect to the integrity of the current most popular virtual platform.

Assertions are generally missing from the initial design of major object-oriented languages with Eiffel being the only exception. A number of projects are addressing the problem of extending the Java environment with assertions [3]. ESC/Java [19] statically detects some programming errors. Nice [14] is a functional Java like language with assertions that are enforced at run-time. JML [27] annotates Java programs with behavioral specifications that are compiled and enforced at run-time, and LOOP [13] generates theories (PVS in particular) representing the semantics of Java classes so that they can be verified by a theorem prover. We are not aware of any research which takes the position that constraints, just like parametric types, also require proper support in the underlying virtual platform. This is a novelty in this paper which considers deep integration of constraints into the overall environment, starting from the language down to different levels of the supporting architecture.

A variety of projects related to persistence in Java have been around for some time [10, 24, 16, 17, 23]. Typical extensions of the virtual platform include read and write barriers for

persistent objects that make use of two unused byte codes in the JVM along with a suitable representation of persistent identifiers and pointer-swizzling techniques. But not a single project in this group has ever considered representation of parametric types and constraints. Both are essential for database technologies that require different collection types equipped with integrity constraints. Very few database research results deal with verification of constraints as presented in [12].

Applications of modern program verification systems, PVS in particular [33], to object-oriented languages have been considered in several papers such as [20, 4]. Unlike Eiffel's run-time checking of assertions, these efforts lead toward static verification. However, we are not aware of any research that considers connecting a sophisticated program verification system with the virtual platform of an object-oriented language. A major point of this paper is that dynamic loading available in languages running on virtual platforms requires load-time verification, hence this connection of a program verification system to the virtual platform is necessary. Completely static verification becomes a particular case of this more general model of verification of constraints.

The notion of behavioral compatibility expressed by behavioral subtyping [8, 28, 35] has been a topic of a fair amount of research reviewed in part in [26]. Our earlier work reports on temporal object-oriented programming techniques [6], and implementing a declarative temporal object-oriented language on top of the JVM [3]. The approach presented in this paper shares some similarities with [34]. Although not object-oriented, research reported in [34] presents a temporal logic view of state changes and representation in PVS that we extend to the object-oriented paradigm.

9. CONCLUSIONS

Object-oriented languages with parametric types and logic-based constraints (assertions) do indeed require proper support in the underlying virtual platform for these high-level language features.

The required features of these extended platforms include representation of parametric types as class files, representation of assertions in class files, construction of class objects with correct actual type signatures along with their associated constraints, an extended loader to load correct type signatures and constraints into the class objects, and reflective capabilities that report parametric (formal and actual) type signatures along with their associated constraints.

This extended virtual platform may be connected to a modern program verification system in order to check constraints and enforce the rules of behavioral subtyping. Since virtual platforms allow dynamic compilation and loading, it must be possible to verify these conditions dynamically upon loading of class objects. The overall result is correct run-time type information and correct dynamic type checking along with verified behavioral subtyping requirements.

Availability of constraints is particularly important in persistent and database systems where yet another extension of

the virtual platform is required to support orthogonal persistence. Both parametric polymorphism and constraints are essential for database technologies that require different collection types equipped with integrity constraints.

Our experimental results show that it is actually possible to extend an existing virtual platform such as the JVM to produce a virtual platform with the above described, much more general desirable capabilities.

10. REFERENCES

- [1] O. Agesen, S. N. Freund, and J. C. Mitchell, Adding type parameterization to the Java language, Proceedings of OOPSLA '97, ACM, pp. 49 -65.
- [2] S. Alagić, S. Kouznetsova, Behavioral compatibility of self-typed theories. Proceedings of ECOOP 2002, *Lecture Notes in Computer Science 2374*, pp. 585-608, Springer, 2002.
- [3] S. Alagić, J. Solorzano, and D. Gitchell, Orthogonal to the Java imperative, Proceedings of ECOOP '98, *Lecture Notes in Computer Science 1445*, pp. 212 - 233, Springer, 1998.
- [4] S. Alagić and J. Logan, Consistency of Java transactions, Proceedings of DBPL 2003, *Lecture Notes in Computer Science 2921*, pp. 71-89, Springer, 2004.
- [5] S. Alagić and T. Ngyen, Parametric polymorphism and orthogonal persistence, Proceedings of the ECOOP 2000 Symposium on Objects and Databases, *Lecture Notes in Computer Science 1813*, pp. 32 - 46, 2001.
- [6] S. Alagić, Temporal object-oriented programming, *Computer Journal*, 43, 6, pp. 491 - 511, 2001.
- [7] S. Alagić, The ODMG object model: does it make sense?, Proceedings of the OOPSLA '97 Conference, pp. 253-270, ACM, 1997.
- [8] P. America, Designing an object-oriented language with behavioral subtyping, *Lecture Notes in Computer Science 489*, pp. 60-90, Springer, 1991.
- [9] M. Atkinson and R. Morrison, Orthogonally persistent object systems, *VLDB Journal* 4, pp. 319-401, 1995.
- [10] M. Atkinson, L. Daynes, M. J. Jordan, T. Printezis, and S. Spence, An orthogonally persistent JavaTM, ACM SIGMOD Record 25, pp. 68-75, ACM, 1996.
- [11] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, and S. Zdonik, The object-oriented database system manifesto, Proceedings of the First Object-Oriented and Deductive Database Conference (DOOD), pp. 40-75, Kyoto, 1989.
- [12] V. Benzaken and X. Schaefer, Static integrity constraint management in object-oriented database programming languages via predicate transformers, Proceedings of ECOOP '97, *Lecture Notes in Computer Science 1241*, pp. 60-84, 1997.
- [13] J. van den Berg and B. Jacobs, The LOOP compiler for Java *Lecture Notes in Computer Science 2031*, Springer, 2001, pp. 299 - 312.
- [14] D. Bonniot, The Nice programming language, <http://nice.sourceforge.net/>.
- [15] G. Bracha, M. Odersky, D. Stoutamire and P. Wadler, Making the future safe for the past: Adding genericity to the Java programming language, Proceedings of OOPSLA 1998, ACM.
- [16] B. Bretl, A. Otis, M. San Soucie, B. Schuchardt, and R. Venkatesh, Persistent Java objects in 3 tier architectures, in: R. Morrison, M. Jordan, and M. Atkinson: *Advances in Persistent Object Systems*, pp. 236-249, Morgan Kaufmann Publishers, 1999.
- [17] R. G. G. Cattell, D. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, and F. Velez, *The Object Data Standard: ODMG 3.0*, Morgan Kaufmann, 2000.
- [18] B. Cabana, S. Alagić and J. Faulkner, Parametric polymorphism for Java: Is there any hope in sight? ACM SIGPLAN Notices, Dec. 2004.
- [19] C. Flanagan, K. R. M. Leino, G. Nelson, J. B. Saxes, and R. Stata, Extended static checking for Java, Proceedings of PLDI, ACM, 2002, pp. 234-245.
- [20] B. Jacobs, L. van den Berg, M. Husiman and M. van Berkum, Reasoning about Java classes, Proceedings of OOPSLA '98, pp. 329-340, ACM, 1998.
- [21] Java Core Reflection, JDK 1.1, Sun Microsystems, 1997.
- [22] Java 5.0, Sun Microsystems, 2004.
- [23] JavaTM Data Objects, JSR 00012, Forte Tools, Sun Microsystems Inc., 2000.
- [24] M. Jordan and M. Atkinson, Orthogonal persistence for Java - A mid-term report, in: R. Morrison, M. Jordan, and M. Atkinson: *Advances in Persistent Object Systems*, pp. 335 - 352, Morgan Kaufmann Publishers, 1999.
- [25] A. Kennedy and D. Syme, Design and implementation of generics for the .NET Common Language Runtime, Proceedings of PLDI, pp. 1 - 12, ACM, 2001.
- [26] G. T. Leavens and K. K. Dhara, Concepts of behavioral subtyping and a sketch of their extension to component-based systems, in: G. T. Leavens and M. Sitaraman, *Foundations of Component-Based Systems*, Cambridge University Press, 2000.
- [27] G. Leavens and Y. Cheon, Design by contract with JML, Iowa State University, 2004.
- [28] B. Liskov and J. M. Wing, A behavioral notion of subtyping, *ACM Transactions on Programming Languages and Systems*, 16, pp. 1811-1841, 1994.
- [29] T. Lindholm and F. Yellin, *The JavaTM Virtual Machine Specification*, Addison-Wesley, 1996.
- [30] B. Meyer, Eiffel: *The Language*, Prentice-Hall, 1992.
- [31] B. Meyer, *Object-Oriented Software Construction*, Prentice Hall, 1997.
- [32] A. Myers, J. Bank, and B. Liskov, Parameterized types for Java, Proceedings of POPL, pp. 132-145, ACM, 1997.
- [33] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Clavert: PVS Language Reference, SRI International, Computer Science Laboratory, Menlo Park, California.
- [34] A. Pnueli and T. Arons, TLPVS: A PVS-based LTL verification system, In: *Verification: theory and Practice, Lecture Notes In Computer Science Vol 2772*, Springer, 2004.
- [35] C. Ruby and G. Leavens, Creating correct subclasses without seeing superclass code, Proceedings of OOPSLA 2000, pp. 208 - 228, ACM, 2000.
- [36] J. Solorzano and S. Alagić, Parametric polymorphism for JavaTM: A reflective solution, Proceedings of OOPSLA '98, pp. 216-225, ACM, 1998.
- [37] The .NET Common Language Runtime, <http://msdn.microsoft.com/net>.