

Reflective Constraint Management for Languages on Virtual Platforms

Mark Royer, Suad Alagić, and Dan Dillon

Department of Computer Science,
University of Southern Maine,
Portland, ME 04104-9300

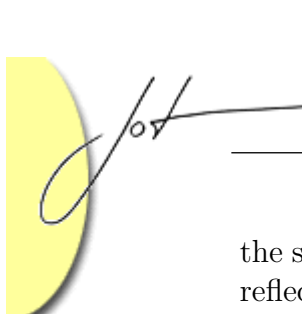
Extending an object-oriented type system with assertions makes it possible for programs using reflection to rely on semantic information to ensure correct use of discovered types. Using extended reflective capabilities to access assertions in (dynamically) loaded class objects allows a variety of general and flexible verification techniques. The XVP (Extended Virtual Platform) implements these features by extending the Java Virtual Machine with the proposed functionalities. Its architecture and applications are described in the paper. One of the goals of the XVP is to provide a virtual platform that supports JML and the programming by contract methodology.

1 INTRODUCTION

Constraints are declarative, logic-based specifications, that are largely missing from major object-oriented languages. Research efforts in several projects [20, 11, 7] have been directed toward overcoming this limitation of object-oriented programming languages. Constraints appear as assertions specified as method preconditions, postconditions, and class invariants in the programming by contract methodology as implemented in Eiffel [23], JML [20], Spec# [7], and OCL [24]. In this form, constraints also appear in the rules of behavioral subtyping [21, 18]. These rules guarantee correct software reuse.

As it is, there is nothing in the underlying Java technology that would guarantee this core property of the object-oriented paradigm. Currently in Java, there are no restrictions as to how a subclass can override an inherited method as long as the typing rules are satisfied. This makes it possible to create semantic (behavioral) incompatibilities between objects of a subclass with respect to objects of its superclass. The availability of constraints makes it possible to enforce behavioral compatibility of a subclass with respect to its superclass. This is a key component of the programming by contract methodology.

Constraints allow the semantics of methods to be expressed in a declarative style. In the current Java technology, accessing classes that were not available at compile time via Java Core Reflection produces type signatures for classes, methods, and fields. But the programmer is in the dark as to what exactly those methods are doing, or what the properties of objects of the discovered classes are. Currently, if



the source code is not available, the semantic information cannot be introspected by reflection, hence correctly using classes discovered by reflection is a very problematic matter. Availability of constraints by reflection provides the basic semantics of classes and methods. This is a prerequisite for their correct use.

Constraints are also critical for reducing the impedance mismatch between data and programming languages. Constraints appear as data integrity conditions in database systems that transactions are required to respect. Major query languages such as SQL and OQL also include constraints, and in fact a query includes a constraint which specifies the collection of objects that qualify for the result. Similarly, the notion of a database transaction cannot be expressed properly because of lack of constraints, and queries cannot be expressed in a declarative style, as in data languages. None of these features are possible in current object-oriented languages. Absence of general constraints is also a major limitation of object-oriented technologies such as ODMG [10] and JDO [16].

Lack of proper support for constraints makes it impossible to have reflective capabilities of a virtual platform that report information on constraints, and on the ability to verify behavioral subtyping requirements. Systems that incorporate constraints typically do so by compiling them into procedural code as in Eiffel [23] and JML [20]. This makes these constraints inaccessible by run-time reflection. If constraints are integrated simply as strings, as in Spec# [7], their structure must be rediscovered when loading class objects, rather than being integrated into the run-time type system as in our extended platform. And in JML constraints are currently not available even in string form.

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. Loading class objects and performing static verification is a preferable mode of operation in the proposed extended platform. However, if the types that were not known at compile time are discovered only at run-time, their associated constraints must then also be verified at run time (more precisely at load time) and actions taken depending upon the outcome of introspection and verification of constraints.

To overcome the problems outlined above, this paper advocates that virtual platforms for object-oriented languages with logic-based constraints (assertions) must provide proper support for this high-level language feature. We present a system called XVP (Extended Virtual Platform) where constraints are managed in their declarative form which is associated with the type information available both statically and dynamically. The typing environment is extended at compile-time and at run-time with semantic information expressed by logic-based constraints.

In spite of the fact that we use the term extended virtual platform, it is critical to point out what kind of an extension this is. This extension requires an elaborate implementation, but it does not affect the integrity of the existing platform in any way. This property is due to the fact that all the essential internals of the platform

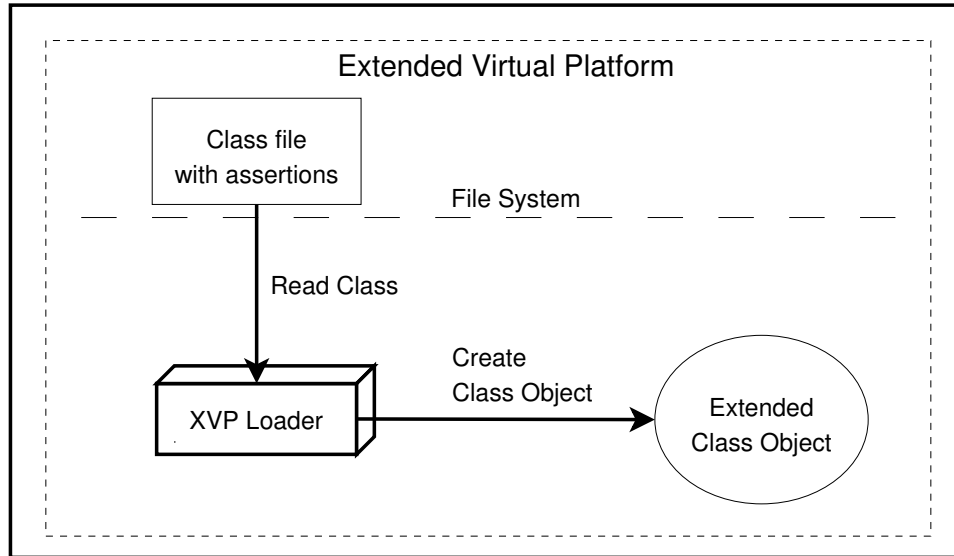


Figure 1: Platform architecture

are extended in such a way that neither portability nor correct execution of legacy code ever becomes an issue. An application-oriented test presented in section 10 demonstrates compatibility with legacy code. In this test, a legacy object-oriented database system is executed on top of the XVP rather than on top of the JVM, and the advantages of constraint management as available via Java Core Reflection in XVP are demonstrated.

2 GENERAL ARCHITECTURE

The general architecture of the extended virtual platform is represented in figure 1. The main components of this architecture are:

- Class files that allow representation of logic-based constraints.
- Class objects that contain type signatures along with constraints.
- A loader that assembles class objects from class files and properly manages type signatures and constraints.
- Reflective capabilities that allow introspection of types and constraints.

Our system is designed in such a way that it is independent of a particular constraint language and its underlying logic basis. However, our project is a part of a greater JML [20] effort. The JML assertion language and its tools play a major role in our development. Although the architecture is more general, this constraint language is used throughout the paper and in the actual implementation.

A tool that interfaces with a program verification system allows checking of constraints and behavioral subtyping requirements. This is represented in figure 2. The program verification system accesses loaded class objects through a tool that

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.

As shown in figure 2, verification techniques for a particular system, PVS (Prototype Verification System [26]), have been investigated; however, interfacing the XVP with other verification systems is also possible.

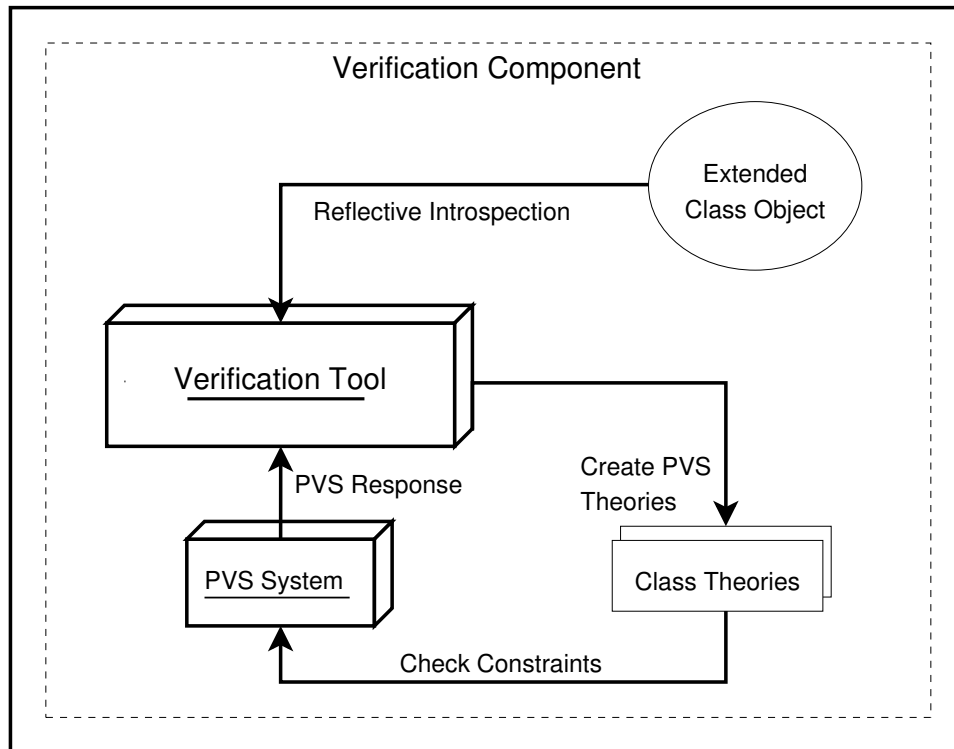


Figure 2: Verification tool

Our system allows a variety of verification techniques. Static verification is possible by loading class objects, accessing them by reflection and carrying out interactive verification typical for the verification system used in this project, PVS. Load time dynamic verification is possible as soon as a class object is loaded and is thus available for introspection by reflection. Enforcing assertions at run-time is based on the fact that the JML compiler generates byte code for constraints. In addition, disciplines other than design by contract (typical for the current JML tools) may be enforced both statically and dynamically because of the availability of assertions that are integrated with the type system. A variety of proof strategies have been developed [5] that may be used as decision procedures. This is in fact required for dynamic verification. Customized verification techniques may thus be developed that are not necessarily based on behavioral subtyping.



3 CONSTRAINTS

We introduce the basic features of the assertion language used in the paper with a sample interface `Collection`. The interface is equipped with JML assertions. Unlike JML, the assertions are specified using the new annotation feature in Java 5.0. This feature is an improvement over JML because it makes assertions available in string format even at run-time. This opens up more general possibilities in managing assertions. Annotations are preceded by `@`.

```
@Constraint("invariant (\\forall Object o; this.contains(o);" +
            "this.count(o) > 0);")
public interface Collection {

    @pure
    public int count(Object o);

    @Constraint("ensures \\result == this.count(o) > 0;")
    @pure
    public boolean contains(Object o);

    @Constraint("ensures this.count(o) == \\old(this.count(o)) + 1 &&" +
            "(\\forall Object o1; this.contains(o1);" +
            "!o1.equals(o) ==> (this.count(o1) == \\old(this.count(o1))))");
    public void insert(Object o);

    @Constraint({ "requires this.contains(o);" ,
            "ensures this.count(o) == \\old(this.count(o)) - 1 &&" +
            "(\\forall Object o1; this.contains(o1);" +
            "!o1.equals(o) ==> (this.count(o1) == \\old(this.count(o1))))" });
    public void delete(Object o);
}
```

Figure 3: Collection interface

The mutator methods `insert` and `delete` that change the underlying object state are equipped with method preconditions and method postconditions specified in the `requires` and `ensures` clauses. The postconditions for the methods `insert` and `delete` refer to both the current and the previous object states. The latter are denoted using the keyword `old` as in Eiffel or JML. The constraints include features from first order predicate calculus, such as quantifiers. Constraints for pure methods that are just functions (do not change the object state) are specified in the same style. The class invariant is also specified.

4 REFLECTIVE CAPABILITIES

The existing Java reflective capabilities allow introspection of type signatures and the extended virtual platform allows introspection of the constraints associated with those types. Constraints are reported by the extended reflective capabilities in their logic-based declarative style. This is a major distinction in comparison with existing virtual platforms.

The Java Core Reflection (JCR) classes that have been extended are `Class`, `Constructor`, and `Method`. These extensions are based on new types such as `Invariant`, `PreCondition`, and `PostCondition`. With these new types it becomes possible to add method preconditions and postconditions to the class `Method`, and the class invariant to the class `Class`. These assertions require further types that make it possible to create objects that represent logical formulas for constraints. In order to achieve independence of a particular constraint language and its logic basis, the types representing logical formulas are specified as abstract classes. These classes must be extended for a particular assertion language as we did for JML.

The structure of constraints is determined by their underlying logic basis. This is why the class `Sentence`, given 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. An example of a sentence is an invariant:

```
(\\forall Object o; this.contains(o); this.count(o) >= 0);
public abstract class Sentence {
    public Variable[] getVariables();
    public Variable[] getExistentialVariables();
    public Variable[] getUniversalVariables();
    public abstract Boolean evaluate(Object[] variables);
}
```

Methods of the class `Variable` report names and types of variables. A logical formula in general contains free variables. An example of a formula is `this.count(o) >= 0`. This is reflected in the definition of the abstract class `Formula`. If the values of free variables are bound to values invoking the method `bindVariables`, a formula may be evaluated. But a specific logic still must be chosen in order to perform the evaluation of a formula.

```
public abstract class Formula extends Sentence {
    public Variable[] getFreeVariables();
    public void bindFreeVariables(Object[] vars);
}
```

The basic building blocks for constructing expressions are terms. An example of a term is `this.count(o)`. 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.

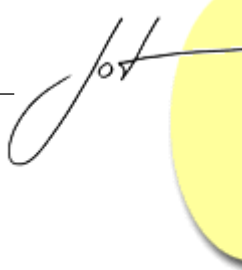


Figure 4: Reflection extensions

```

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 abstract 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.

Some of the basic extensions to Java Core Reflection are shown in figure 4. Additions of the recompiled class `Class` allow access to the declared and the inherited invariant.

```

public final class Class { ...
    public Invariant getInvariant();
}
  
```

The extensions of the class `Method` allow access to (declared and inherited) preconditions and postconditions. The class `Constructor` is similarly extended.

```

public final class Method { ...
    public PostCondition getPostCondition();
    public PreCondition  getPreCondition();
}
  
```

5 REPRESENTATION OF CONSTRAINTS

Java classes are compiled into Java class files. In the Java class file structure [22] an array of attributes is associated with each class, each field, and each method. The predefined attributes such as `Code`, `ConstantValue`, `Exceptions`, etc., must be correctly recognized in order for the Java Virtual Machine to function correctly. An important point is that in addition to the above mentioned attributes, optional attributes are also allowed. These optional attributes have no effect on the correct functioning of the Java Virtual Machine.

In the XVP a class has an invariant, and each method of a class contains a precondition and a postcondition. The structures representing these assertions are assembled by an extended loader from the optional attributes contained by the class file. This is accomplished by constructing invariants, preconditions, and postconditions, lazily, i.e., upon demand via Java Core Reflection. This is a very different approach to managing constraints in comparison with other systems.

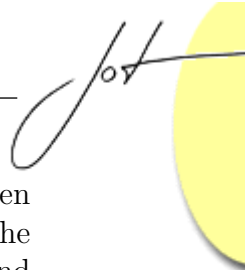
In order to enforce semantic conditions, projects such as JML [20] and Octopus [25] are forced to create complex techniques for representation of assertions in procedural form. For example, when JML's runtime assertion checker is used, interfaces are extended with surrogate inner classes and class definitions are extended with extra methods that were not written by the programmer. When these types are introspected via Java Core Reflection, the additional constraint-related implementation information unexpectedly appears intermingled with the type information the programmer would expect. Consequently, this means that the runtime representation of the type declared by the programmer does not match the type the programmer intended.

Unlike the above mentioned projects, the XVP makes small extensions to the classes `Class`, `Method`, and `Constructor`. In these extensions type information is recorded as usual by Java Core Reflection; however, additional extensions are made to record, but keep separate, semantic information with the corresponding types. This technique is similar to the technique used to extend Java with generics in Java 5.0 and is further elaborated in section 7.

The representation of constraints in class files is based on optional attributes as specified in the JVM Specification [22]. Attached to the class file structure is an optional attribute representing an invariant. Similarly, method structures in the class file have optional attributes representing preconditions and postconditions.

The assertion attributes are further composed of different structures. Some of these structures represent terms and others represent formulas based on those terms. The structure of formulas is determined by the logic underlying the constraint language.

Term structures are designed in such a way that they can represent five possible types of terms: messages, constants, variables, new object terms, and field access terms.



A formula may be an atom or a complex formula. If the formula is an atom, then it refers to a message term, and the descriptor in the message term will indicate the result type `boolean`. A formula contains arrays of free, universally quantified, and existentially quantified variables. Each variable is represented by a pair, consisting of the type and the name of the variable. Connective types in formulas are meant to be one of the standard ones (and, not, or, implies, etc.).

6 COMPILING DECLARATIVE SPECIFICATIONS

The procedure for compiling Java source equipped with JML assertions is represented in figure 5 and it consists of the following actions:

- First, Java source equipped with constraints is compiled with the Java 5.0 compiler.
- The next step is the XVP post compiler which performs limited processing of the source. It gathers the information from the import clauses in the source and the signatures of formal parameters (names and types) of methods and constructors. It collects the information in each annotated assertion and sends it to the module that performs parsing and type checking of the assertion. This is performed for every class, method, and constructor.
- The subsequent step is creating the class file byte code representation of assertions. This is done in accordance with the class file format for constraints discussed in section 5. The class file augmented with constraints actually overwrites the class file generated by the Java 5.0 compiler.

Java 5.0 allows annotations to be accessed at compile time, load time, or at run time using Java Core Reflection. JML currently converts assertions into executable code at compile time. Our approach allows the original source of an assertion to be available at runtime. This makes it possible to compile assertions at runtime if they were not processed statically by the XVP post-compiler. The `@Constraint` annotation is represented by the following declaration:

```
@Target({ElementType.METHOD,ElementType.TYPE,ElementType.CONSTRUCTOR})
@Retention(RetentionPolicy.RUNTIME)
public @interface Constraint {
    String[] value();
}
```

The `Target` meta-annotation indicates that the `Constraint` annotation applies to types, methods, and constructors. The `Retention` meta-annotation indicates that this information is to be retained at runtime. The `Constraint` annotation has a `String` array named `value`.

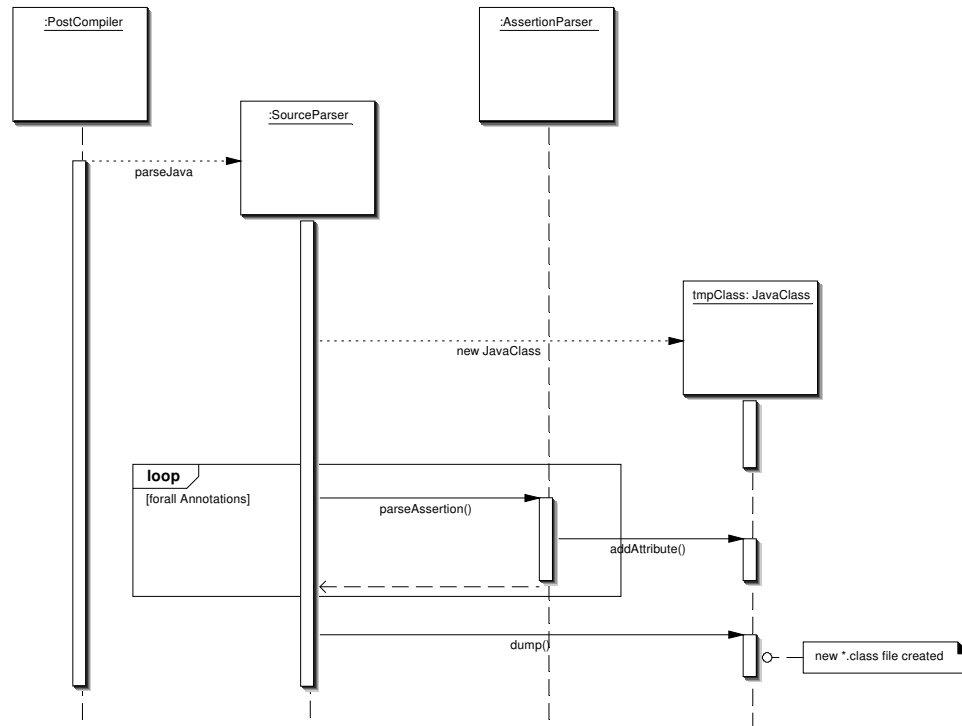
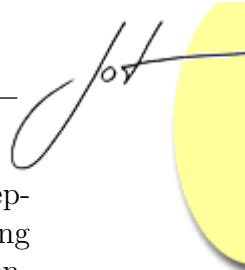


Figure 5: Compiler sequence

Once class files include assertions this way, they may be interrogated at runtime for information about the assertions. The choice of representing assertions as annotations is based on the flexibility of this approach, which allows assertions to be parsed from raw `String` form at compile time, or dynamically at load time.

A limited post-compile parse of the Java source file is required because the Java compiler discards import statements and formal parameter names during compilation. If this parse were omitted, every name in a JML expression would need to be fully qualified (i.e., `Class` would need to be specified as `java.lang.Class`), and more importantly, specifications that refer to formal parameters of methods and constructors could not be bound correctly without more information. In addition, there would be limitations on dynamically parsing assertions. Preconditions and postconditions could not refer to parameter names, and every assertion would need to use fully qualified names to compile.

At each new type encountered in the source file parsing phase BCEL [6] is utilized to store information regarding assertions. BCEL (Byte Code Engineering Library) is part of the standard Java runtime environment. A temporary class file of type `JavaClass` is created. `JavaClass` is part of BCEL and it represents all the class file information of a class annotated with assertions. The `JavaClass` object also provides methods to add, remove, and modify features of a class file it is representing.



At each annotation of type `Constraint`, the annotation contents (strings representing the constraints to be parsed) are sent to a separate assertion processing module, along with the current class file, import declarations, methods and constructors, and their formal parameter names and types.

The assertion processing module is responsible for verifying that annotation content is valid with respect to the JML grammar for assertions. There is a very important reason why this module is separate from the module which parses the Java source file. Our design allows any constraint language to be substituted in place of our current module which validates JML expressions. A different constraint language would require a different module for verifying syntactic correctness of assertions, but the rest of the architecture will remain unaffected by this change of logic. An example of a different constraint language would be OCL [24].

The challenging aspect of type checking assertions that are integrated into the Java type system, has been parsing declarative expressions in a language very different from Java, and verifying their correctness using the procedural tools that Java provides. While resolving types, our approach is nearly identical to how Java 5.0 resolves names, but we have to handle much more general situations such as expressions with quantified variables as defined in the JML grammar.

Our approach is also different from the Java compiler's approach because we extensively use Java Core Reflection to infer types. Since all necessary class files have already been generated by the Java compiler and the corresponding class objects have already been loaded, it is easy to discover type information via Java Core Reflection. This explains why Java Core Reflection is used for discovering type information when performing type checking.

7 CONSTRAINT MANAGEMENT

In this section we describe how constraints are managed in the extended virtual platform. Extensions to existing components of Java Core Reflection are made in such a way that they allow for a variety of constraint languages to be represented and maintained by the system.

The class `ConstraintManagement` is in charge of associating constraints with their corresponding types. There are several components that are utilized for managing assertions via the `ConstraintManagement` class in the XVP. The main ones are:

- An object repository
- A constraint factory
- A byte code repository
- An XVP constraint representation
- A JML constraint representation

In order to allow users to quantify over all currently existing objects, an object repository is used to maintain references to every object that is created during

program execution on the virtual platform. This kind of quantification is perfectly natural and it is not supported in JML.

This is accomplished by extending the default constructor in the class `Object` so that the newly created object is added to the object repository. This is required because quantifiers may refer to all objects of a given type, and the current JVM does not keep track of that collection. This represents a generalization of both JML and JVM. In the object repository, objects are kept in maps according to their types to allow for constant time access to any object in the repository.

References to objects in the repository are weak references. This allows objects to be garbage collected when they are no longer referenced in applications running on the system. When an object is garbage collected, the weak reference in the repository must be removed. This is accomplished by the garbage collector adding newly destroyed weak object references to a list of dead references, and then the repository can remove the weak references at appropriate times.

A `ConstraintFactory` is used by the constraint manager to create assertions requested by the system. The `ConstraintFactory` is an abstract class that follows an abstract factory pattern design. This allows the constraint management class to dynamically load bytes from the byte repository when an assertion is requested and lazily instantiate it with a specific constraint factory. The newly created assertion is then handled by the constraint manager and passed back to the corresponding reflection class, either `Class`, `Method`, or `Constructor`.

In order for the `ConstraintManagement` and extended Java Core Reflection classes to handle a variety of constraint languages and their different logic bases, an abstract representation of the components of a constraint language is used. The types of terms such as `VariableTerm`, `MessageTerm`, `FieldTerm`, etc., and abstract classes such as `Sentence` and `Formula` form an abstract framework. This framework is extended with a concrete representation for each specific constraint logic.

In our design, we extended the abstract logic representation with the specific one according to the grammar of JML. This was accomplished by creating concrete classes that represent the various types of JML expressions and a specific constraint factory that knows how to properly construct JML expressions. Each abstract representation of formulas and terms was extended by its corresponding component according to the JML grammar.

Figure 6 shows the associations between the `ConstraintManagement` class and the components described above.

Classes are loaded via the extended `XVPLoader`, and the byte code representation of assertions are stored in the `ByteCodeRepository`. When a class or method of Java Core Reflection requires an assertion, it requests for the assertion from the `ConstraintManagement` class. `ConstraintManagement` gets the corresponding bytes from the `ByteCodeRepository` and then creates the assertion using the `ConstraintFactory`. At runtime, the `ConstraintFactory` is represented by a concrete factory that knows how to create assertions of the underlying specific assertion

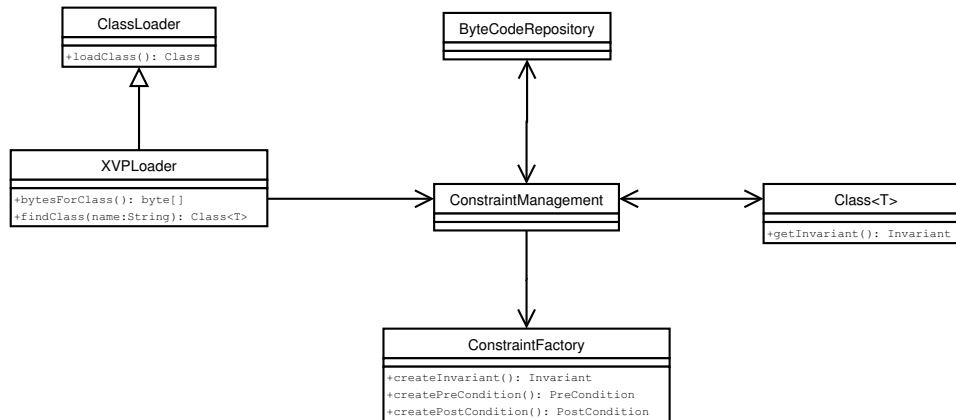


Figure 6: Main constraint components

language.

The XVP must carefully manage expressions in assertions that use the keyword `old`. In the XVP, the value of the argument expression `e` in `old(e)` is stored prior to method execution so that it can be retrieved after method execution. Primitive and object types are handled differently by the XVP. Prior to method execution, a primitive value is stored by the XVP. If the type of the expression `e` is an object type, instead of a copy of that object, only a reference to an object is saved. This means that given an object `obj`, `old(obj) == obj` after method execution.

A further complication in managing the occurrence of `old` in expressions is when it appears in the scope of collection operations as defined in JML [20]. The XVP supports seven collection operations: `forall`, `exists`, `max`, `min`, `product`, `sum`, and `num_of`. In general, a collection operation is composed of three parts: a collection operator followed by a type declaration, a range predicate, and a body predicate. This is illustrated by the following example:

```
(\forall Object o; this.contains(o); this.count(o) >= 0);
```

For primitive types in collection operations, the XVP stores values using a mapping scheme that allows for nested quantifiers and retrieval of `old(e)` values after method execution. A similar system is used by the XVP for object types, but in this case the XVP may throw a `NoOldValueException`. `NoOldValueExceptions` can be thrown in collection operations because collection operations in the XVP may range over all object types. This may include newly created objects that did not exist in the previous system state.

8 EXTENDED LOADER

As shown in figure 6, an important part of managing constraints in the extended virtual platform is an extended class loader. The virtual platform is extended to be able to properly load constraints in extended Java class files. There are two

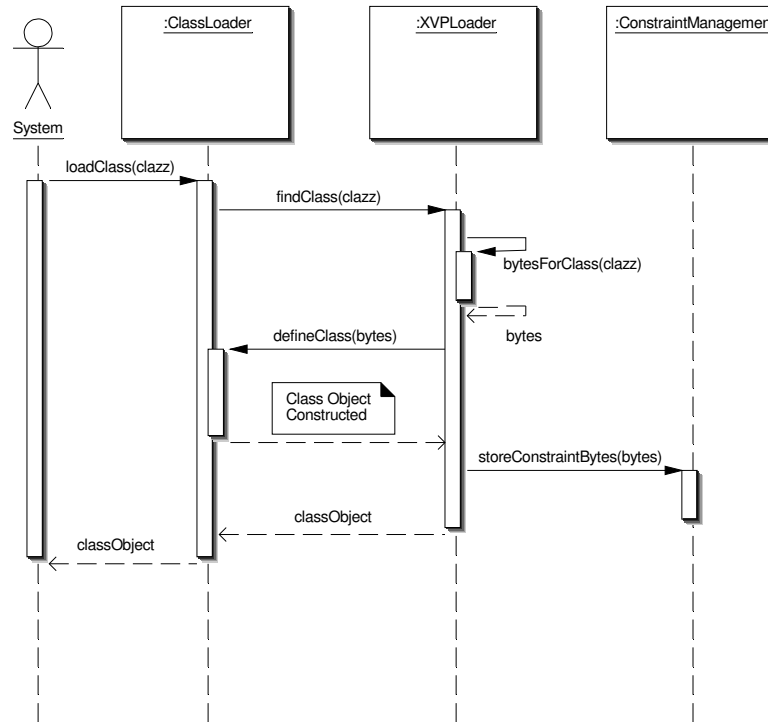
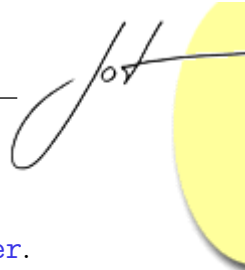


Figure 7: Class loading sequence

design choices here. The first one is to extend the native system class loader, and the second is to utilize Java's parent delegation class loader model and create an extended class loader.

In the current implementation of XVP the extended class loader option is used. The reason is that it allows us to load all augmented class files of interest without changing any native code. This is useful because it allows us to port our system to a variety of virtual machines without having to recompile the entire platform. The reason this is appropriate is that the Java API classes are not compiled with assertion declarations. Since the default system class loader is in charge of loading the system runtime library classes, if these files were to be annotated with assertions, then the system class loader would need to be extended. In addition, the XVP does not have access to assertions in classes that are loaded in systems that define their own class loaders. The XVP implements a null object pattern for assertions, so assertions in classes loaded by other loaders are treated as always `true`.

The general loading process for the XVP is shown in the sequence figure 7. The loading process begins when the XVP requires a class object with associated constraint information. At this point `loadClass` is invoked on the parent class loader. When the default system loader fails to find the class in the system class path, the overridden `findClass` method is invoked on the `XVPLoader`. The `XVPLoader` has enough information to locate the augmented class file with assertions in the file system. This occurs in the method `bytesForClass`, and the byte code representation



of the class file is returned after loading the class file from the file system.

Two actions are now taken on the class byte representation by the `XVPLoader`. First, the `XVPLoader` finishes creating the class object by handing the byte code representation to the `defineClass` method in `ClassLoader`. The `defineClass` method ensures that the byte code representation of a class is properly formed and returns a completely loaded class object. Secondly, the `XVPLoader` passes the byte code representation of assertions to the `ConstraintManagement` class where the byte code is stored for future lazy instantiation. Assuming that there were no complications during the loading process, the system now has a class object with associated constraints ready for use in an application program.

The components that allow access to assertions in extended class files are the `XVPLoader` and BCEL. Special attention is made to class paths to ensure that the `XVPLoader` is selected as the class loader of classes with assertion information. After the class files have been loaded as a byte code representation, the optional attributes containing constraint information have to be recognized and handled by the `XVPLoader`. This is done using BCEL. The byte code representation of assertions is stored by the `ConstraintManagement` class for access by the program application if needed.

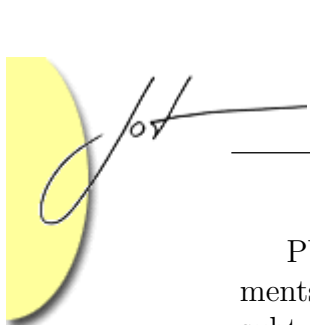
9 VERIFICATION SYSTEM

A distinctive feature of the extended virtual platform is the representation and management of constraints in their logic-based, declarative form, rather than in the procedural form, as in Eiffel or JML. This makes it possible to interface the platform with a verification system using extended reflection.

PVS [26] is a specific verification system that we have been investigating [4]. 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. These features allows representation of generic classes as in Java 5.0 and a variety of constraint languages.

The first order predicate calculus-based constraints appear in the sample class `Collection` given in section 3. In fact, we use a temporal logic to express the subtleties of the object-oriented paradigm such as object-state changes caused by mutator methods that require usage of the operator `old` [4]. This makes it possible to specify history properties as defined in [21]. These are properties of sequences of object states, pairs of successive object states in particular. History properties are available in JML via the usage of the `old` operator in assertions.

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. The XVP reflection extensions use a visitor pattern to facilitate this transformation. 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 [4].



PVS can be used in the extended platform to verify behavioral subtyping requirements [21]. When a class is loaded, PVS can check 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. Stronger forms of behavioral subtyping are based on history properties and require compatibility of sequences of object states [21]. These requirements are represented in temporal logic based verification techniques described in [4]. In addition, unlike the situation in the current version of JML and Eiffel, other policies for enforcing constraints may be used. Full details are given in [4].

The compromise between static and dynamic checking amounts to treating constraints that are meant to be checked at run-time as axioms and proving the remaining ones as theorems. The proofs will be valid as long as the truth of the axioms is guaranteed at run-time. At the same time, the constraints that are proved as theorems under the above assumptions will not be checked at run-time. This improves efficiency of dynamic checking of constraints and reliability of programs.

10 ILLUSTRATIVE APPLICATION

Our application illustrating the advantages offered by the extended virtual platform is from the database area where constraints are absolutely critical. In spite of that, practically all object-oriented database technologies (ODMG [10] and JDO [16] in particular) are lacking general, logic-based constraint capabilities. Yet, constraints are available in other data models such as relational or XML (represented by XML Schema).

Extended reflection is critical because database users expect to see declarative constraints when introspecting classes in a database schema. Procedural representation of constraints as in Eiffel or JML is completely unacceptable for introspection. Static verification of database transactions is an attractive goal that we advocate, but in reality dynamic verification is unavoidable. This is why more general and more flexible techniques for managing and verifying constraints, available in the XVP, represent a significant advantage.

JML and Eiffel were designed to support design by contract, but database properties that are typically enforced are not behavioral subtyping, but rather constraints such as keys, referential integrity, range values, sums, etc. All of this shows that more general and more flexible constraint management is needed in comparison with what is currently offered for object-oriented languages.

In this experiment, we used Versant's FastObjects [27], an ODMG object-oriented database management system. Just like ODMG, FastObjects does not know anything about constraints. Since FastObjects does not have such a capability, we construct a transaction example below showing how a particular transaction can access assertions using extended reflection, evaluate them, and proceed depending



upon the result, i.e., abort or commit.

```
public void update() {
    Transaction txn = new Transaction();
    txn.begin();
    Employee employee = null;
    try { employee = (Employee)database.lookup("John Doe");
    } catch (ODMGException e) { txn.abort(); /* other actions */ }

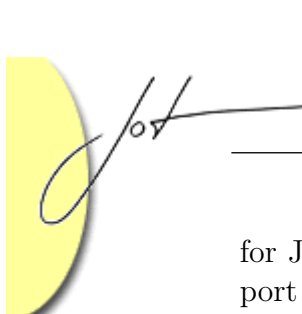
    Invariant inv = employee.getClass().getInvariant();
    Method m = null;
    try { m = employee.getClass().getMethod("updateSalary", float.class);
    } catch (Exception e) { txn.abort(); /* other actions */ }
    PreCondition pre = m.getPreCondition();
    Object[] params = new Object[]{1000f};
    if (! (pre.evaluate(employee, params) && inv.evaluate(employee)) )
        txn.abort();
    PostCondition post = m.getPostCondition();
    post.bindPreMethodVars(employee, params);
    employee.updateSalary(1000f);
    if (! (post.evaluate(employee, null, params) && inv.evaluate(employee)))
        txn.abort();
    txn.commit();
}
```

Figure 8: Database transaction

In the example 8 the extended platform has been used with FastObjects rather than the standard JVM. First, the program initializes access to the database. The employee object “John Doe” is looked up in the database and returned. At this point components of the XVP are utilized. The invariant and the precondition are accessed from the employee class and evaluated. If the invariant and precondition evaluate to true, then the program binds variables in postconditions, employs an evaluation strategy that takes into account the semantics of the `old` operator, and continues to execute the update on the employee. After the update has taken place, the postcondition and invariant are evaluated. Given each assertion component evaluates to true, the transaction is completed and committed to the database.

11 RELATED RESEARCH

Assertions are generally missing from the initial design of major object-oriented languages with Eiffel [23] being the only exception. ESC/Java [11] statically detects some programming errors. Nice [9] is a functional Java like language with assertions that are enforced at run-time, and Spec# [7] is a superset of C# equipped with assertions. JML [19] annotates Java programs with behavioral specifications that are compiled and enforced at run-time. LOOP [8] (with a related work [13]) generates theories (PVS in particular) representing the semantics of Java classes so that they can be verified by a theorem prover, and we develop temporal verification techniques



for Java-like classes in PVS [4]. Our position is that assertions require proper support in the underlying virtual platform [3]. 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.

Spec# specifications are both compiled and attached to executable code as custom attributes accessible by CLR meta data capabilities [7]. Unlike Spec# in which these specifications are strings, in our architecture internal representation is produced that is completely integrated into the run-time type system and accessible in a declarative form along with the type information using an extended JCR. In the Spec# architecture the static program verifier Boogie consumes the compiled code. It constructs a program in its own intermediate language and makes use of the Simplifier theorem prover [7]. In our architecture type signatures with associated constraints are visible in class and method objects that are transformed into PVS theories, and PVS is invoked to verify the desired properties.

This project makes several contributions to the existing JML related results. Currently, JML compiles assertions into code that appears in class files and thus makes dynamic checking of assertions possible. In our architecture, in addition to the compiled code for assertions, internal declarative representation integrated with class and method type signatures is also generated so that it is accessible by JCR. This feature is called specification reflection in [17]. In addition, our architecture allows other possibilities discussed in [17] such as reflective specification execution, i.e., execution of assertions discovered at run-time. Customized run-time checking is also possible based on run-time analysis of assertions available via JCR. Since various components of assertions are in fact instances of classes such as variable, term, message, formula, and sentence, even run-time construction of assertions is also possible in our architecture.

12 CONCLUSIONS

Integration of logic-based constraints into an object-oriented programming environment requires proper support. Integrating constraints into the type system makes the constraints accessible by reflection. Programs using reflection can now rely on the semantic information associated with types in order to ensure correct usage of classes and methods discovered by reflection.

The required features of an extended platform include representation of constraints in class files and class objects, an extended loader to load constraints into the class objects, and reflective capabilities that report type signatures along with their associated constraints.

Perhaps the most important point is that all of this can be accomplished with pure extensions of the existing platform without any effect on the integrity or correct functioning of the legacy platform. Even the critical components of the legacy platform such as the structure of class files and objects is not affected. The only extension visible by users are additional types related to constraints that are available



through Java Core Reflection.

A distinctive feature of this environment is that it can be interfaced with a sophisticated program verification system which makes it possible to check constraints and enforce the rules of programming by contract or other rules. The extended reflective capabilities play a decisive role in making this possible. Construction of program verification theories is based on introspecting both type signatures and their associated constraints using reflection.

This architecture is general enough to permit a variety of constraint languages and their logic bases. It also allows a range of verification techniques, from the static to the dynamic ones, as well as various intermediate options. Although programming by contract (or behavioral subtyping) has been our primary interest, other disciplines based on the availability of the semantic information expressed by constraints are also possible. Since constraints can be accessed by reflection, much more general and flexible verification techniques are available to application programs.

WEB SITE

The web site of this project: <http://www.cs.usm.maine.edu/~alagic/xvp> contains documentation, illustrative examples and download information.

REFERENCES

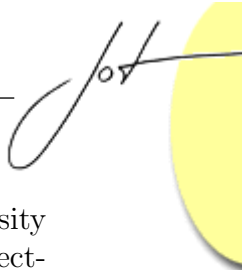
- [1] S. Alagić, S. Kouznetsova, Behavioral compatibility of self-typed theories, Proceedings of ECOOP 2002, *Lecture Notes in Computer Science 2374*, pp. 585-608, Springer, 2002.
- [2] S. Alagić, J. Solorzano, and D. Gitchell, Orthogonal to the Java imperative, Proceedings of ECOOP '98, *Lecture Notes in Computer Science 1445*, pp. 212 - 233, Springer, 1998.
- [3] S. Alagić and M. Royer, Next generation of virtual platforms, <http://www.odbms.org/>, 2005.
- [4] S. Alagić, M. Royer, and D. Crews, Temporal verification theories for Java-like classes, ECOOP 2006 Workshop Formal Techniques for Java-like Programs, <http://www.disi.unige.it/person/AnconaD/FTfJP06/>, 2006.
- [5] M. Archer, B. Di Vito, and C. Munoz, Developing user strategies in PVS: A tutorial, Proceedings of STRATA 2003.
- [6] Byte Code Engineering Library. <http://jakarta.apache.org/bcel/>.
- [7] M. Barnett, K. R. M. Leino, and W. Schulte, The Spec# programming system: an overview, Microsoft Research 2004. Also in Proceedings of CASSIS 2004.
- [8] J. van den Berg and B. Jacobs, The LOOP compiler for Java *Lecture Notes in Computer Science 2031*, Springer, 2001, pp. 299 - 312.
- [9] D. Bennett, The Nice programming language, <http://nice.sourceforge.net/>.
- [10] 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.

- [11] 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.
- [12] J. Gosling, B. Joy, G. Steele and G. Bracha, The JavaTM Language Specification, Addison-Wesley, 2006.
- [13] 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.
- [14] Java Core Reflection, JDK 1.1, Sun Microsystems, 1997.
- [15] Java 5.0, Sun Microsystems, 2004.
- [16] D. Jordan and C. Russell, *Java Data Objects*, O'Reilly, 2003.
- [17] Y. Cheon, Y. Hayashi, and G. Leavens, A thought on specification reflection, Proc. of 8th World Multi Conf. on Systems, Computing Techniques, 2004.
- [18] 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.
- [19] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok, How the design of JML accommodates both runtime assertion checking and formal verification, *Science of Computer Programming*, Vol. 55, pp. 185-205, Elsevier, 2005.
- [20] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cook, P. Muller, and J. Kiniry, JML Reference Manual (draft), July 2005, <http://www.cs.iastate.edu/~leavens/JML/>.
- [21] B. Liskov and J. M. Wing, A behavioral notion of subtyping, *ACM Transactions on Programming Languages and Systems*, 16, pp. 1811-1841, 1994.
- [22] T. Lindholm and F. Yellin, *The JavaTM Virtual Machine Specification*, Addison-Wesley, 2000.
- [23] B. Meyer, *Object-Oriented Software Construction*, Prentice Hall, 1997.
- [24] Object Constraint Language Specification, OMG documents ad970808 and ad2003-01-06.
- [25] Octopus: OCL Tool for Precise UML Specifications, <http://www.klasse.nl/octopus/>.
- [26] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Clavert: PVS Language Reference, SRI International, Computer Science Laboratory, Menlo Park, California.
- [27] Versant FastObjects, <http://www.versant.com/products/fastobjects>.

ABOUT THE AUTHORS



Mark Royer is a PhD student and research assistant in The Department of Computer Science at The University of Maine. His research interests include object-oriented programming languages and database systems. He is currently working on data integration problems at The University of Maine. He can be reached at mroyer@cs.umaine.edu. See also <http://cs.umaine.edu/~mroyer>.



Suad Alagić is a Computer Science Professor at the University of Southern Maine in Portland. His research areas are object-oriented systems, database systems, and programming languages and systems. His object-oriented publications are related to the Java technology, object-oriented persistent and database technology (ODMG in particular), and constraint (assertion) languages. Most of his research has been directed toward integration of the technology of programming languages and systems and database systems. He can be reached at alagic@cs.usm.maine.edu. See also <http://cs.usm.maine.edu/~alagic>.



Dan Dillon is a software engineer in the industry. He is currently developing applications based on Java Enterprise technologies. His interests include Web Services and web-oriented databases. He can be reached at dan.j.dillon@gmail.com.