Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 240 (2009) 5–21
www.elsevier.com/locate/entcs

A Case Study in JML-Assisted Software Development

Néstor Cataño[1]

Department of Mathematics and Engineering
University of Madeira
Funchal, Portugal

Fernando Barraza[2]

ParqueSoft
Calle 25 No. 127-220 Autopista Cali-Jamundí
Cali, Colombia

Daniel García and Pablo Ortega and Camilo Rueda[3,4,5]

Department of Computer Science
Pontificia Universidad Javeriana
Cali, Colombia

Abstract

This paper presents a case study in formal software development of a plugin for a Java Desktop project management application using JML. Our goals for the case study include determining how JML-based formal methods can be incorporated in traditional software engineering practices used in the software industry and how the use of JML for modeling software requirements can enforce the programming of correct Java code. We demonstrate how JML-based formal methods can be used so as to effectively contribute to the making of decisions within a software development team.

Keywords: Java, Java Modeling Language (JML), Formal Methods, Software Development, International Center for Tropical Agriculture (CIAT)

[1] Email:ncatano@uma.pt
[2] Email:fbarraza@parquesoft.com
[3] Email:dagarcia@puj.edu.co
[4] Email:paortega@puj.edu.co
[5] Email:crueda@puj.edu.co

doi:10.1016/j.entcs.2009.05.042

1 Introduction

Software systems have become a key part of all our lives. Everyday examples include bank transaction systems, web-based systems to buy services over the Internet and smart cards running small programs. These systems become more important as they become more pervasive and suppliers and users are both increasingly worried about their correctness. Although software engineering methods provide a disciplined approach to software development, it is still quite common to find flawed software systems. An approach to tackle the problem of constructing correct programs is through the use of mathematical formalisms and mathematically based tools as part of software engineering practices. Formal specifications allow the capture of requirements unambiguously as part of a software engineering methodology. A formal specification can be used to generate a collection of documents describing the expected behavior of a system. This documentation can be used to resolve any differences regarding the expected behavior of the system between members of the quality assurance team, the programmers and the client.

An interesting effort towards the development and use of tools based on a common specification language is JML (the Java Modeling Language) [11,10], which has become the standard language for formally specifying the behavior of Java classes. JML makes it possible to use run-time and static checkers for testing program correctness. Typically, run-time checkers require that a program is transformed into one that includes the checks. Static checking complements run-time checking by automatically attempting to apply standard program verification techniques. In this way, potential run-time problems can be found at compile-time.

For the case study described in this paper, we collaboratively formed a group of software engineers and JML experts for developing a project manager plugin for Ax-LIMS, a Java Desktop web-based application based on LIMS (Laboratory Information Management System), a project management application previously developed in PHP. The software development team was composed of a project manager and 3 experienced Java software developers working at ParqueSoft [15], the biggest Colombian technological cluster, gathering more than 200 software companies and 800 software developers. The JML experts group was composed of one experienced JML researcher, one researcher with experience in using B as a modeling and implementation methodology [1] and two bachelor in computer science undergraduate students from *Pontificia Universidad Javeriana*. The students had some previous exposure to JML and formal methods. The ParqueSoft engineers used the Unified Process (UP) as software methodology for developing the plugin and UML (Unified Modeling Language) [16] as modeling language. However, OCL (Object Constraint Language) was not used as language for expressing constraints on the UML models. Model constraints were expressed in JML instead.

Our goals for the case study included determining how JML-based formal methods can be incorporated in traditional software engineering practices used in the software industry - how software requirements of a commercial application can be modeled in a formal specification language such as JML. We employed the JML common tools [10,2], a suite of tools providing support to run-time assertion checking,

for testing our specifications. JML was found to be expressive enough to formalize the informal software requirements of the Ax-LIMS project manager plugin. Most of these requirements precisely describe the states for projects and tasks managed by Ax-LIMS. The initial version of the code written by the ParqueSoft engineers contained about 40 programming errors which the JML experts group reported. This allowed the engineers to write a correct version of their code. The errors were mostly due to the use of automatic code generators that led engineers to introduce code with bad programming practices. Additionally, we believe that enforcing the writing of correct code as described in Section 5.1 is a major contribution of our work that seeks to make (JML-based) formal methods more popular in software industry. In particular, expressing software requirements as invariants in JML and systematically using formal methods tools for checking the correctness of the code as it is written forces programmers to think about how the written code affects the consistency and the correctness of the whole application.

In the following, Section 2 introduces JML and describes how run-time assertion checking with JML is performed. Section 3 gives an overview of Ax-LIMS and presents the informal software requirements of the Ax-LIMS project manager plugin. Section 4 describes the style of formal specification used for developing the plugin. Section 5 describes how the informal software requirements of the Ax-LIMS project manager plugin are formally specified in JML. This section further presents interesting aspects in the specification and development of the plugin. Finally, Section 6 presents future work and Section 7 gives conclusions.

2 JML

2.1 The Specification Language

JML is a specification language for Java that provides support for B. Meyer's design-by-contract principles [14]. The idea behind the design-by-contract methodology is that a contract between a class and its clients exists. The client must guarantee certain conditions, called pre-conditions, to be able to call a method of the class. In return, the class must guarantee certain conditions, called post-conditions, that will hold after the method is called. JML was started by Gary Leavens and his team at Iowa State University, but is now an academic community effort with many people involved through the development of tools providing support for the language [6,7,8,4,17].

JML specifications use Java syntax, and are embedded in Java code within special marked comments `/*@ ... @*/` or after `//@`. A simple JML specification for a Java class consists of pre- and post-conditions added to its methods, and class invariants restricting the possible states of class instances. Specifications for method pre- and post-conditions are embedded as comments immediately before method declarations. JML predicates are first-order logic predicates formed of side-effect free Java boolean expressions and several specification-only JML constructs. Because of this side-effect restriction, Java operators like `++` and `--` are not allowed in JML specifications. JML provides notations for forward and backward logical

implications, `==>` and `<==`, for non-equivalence `<=!=>`, and for logical *or* and logical *and*, `||` and `&&`. The JML notations for the standard universal and existential quantifiers are `(\forall T x; E)` and `(\exists T x; E)`, where `T x;` declares a variable `x` of type `T`, and `E` is the expression that must hold for every (some) value of type `T`. The expressions `(\forall T x; P; Q)` and `(\exists T x; P; Q)` are equivalent to `(\forall T x; P ==> Q)` and `(\exists T x; P && Q)` respectively.

JML provides specifications for several mathematical types such as sets, sequences, functions and relations. JML specifications are inherited by subclasses – subclass objects must satisfy superclass invariants, and subclass methods must obey the specifications of all superclass methods that they override. This ensures behavioral sub-typing – a subclass object can always be used (correctly) where a superclass object is expected.

In the following, we briefly review JML specification constructs. The reader is invited to consult [12] for a full introduction to JML.

`requires P`. specifies a method pre-condition `P`, which must be `true` when the method is called. Predicate `P` is a valid JML predicate.

`ensures Q`. specifies a normal method post-condition `Q`. It says that if the method terminates in a normal state, *i.e.*, without throwing an exception, then the predicate `Q` will hold in that state. Predicate `Q` is a valid JML predicate.

`signals (E e) R`. specifies an exceptional method post-condition `R`. It says that if the method throws an exception `e` of type `E`, a subtype of `java.lang.Exception`, then the JML predicate `R` must hold. Predicate `R` is a valid JML predicate. JML allows the use of the alternative clause `exsures` for `signals`.

`normal_behavior`. specifies that if the normal method pre-condition holds in the pre-state of the method, then it will always terminate in a normal state, and the normal post-condition will hold in this state.

`exceptional_behavior`. specifies that if the method exceptional pre-condition holds in the pre-state of the method, then it will always terminate in an exceptional state, throwing a `java.lang.Exception`, and the corresponding exceptional post-condition will hold in this state.

`assignable L`. specifies that the method may only modify location `L`. Any other location not listed in `L` may therefore not be modified. This must be true for both normal and exceptional post-conditions. Two special `assignable` specifications exist, `assignable \nothing`, which specifies that the method modifies no location, and `assignable \everything`, which specifies that the method may modify any location. JML allows the use of the alternative clauses `modifies` and `modifiable` for `assignable`.

`\old(e)`. refers to the value of the expression `e` in the pre-state of a method. This specification can only be used in a normal or exceptional method post-condition specifications.

`\fresh(e)`. says that `e` is not null and was not allocated in the pre-state of the method.

`\result`. represents the value returned by a method. It can only be used in a normal or an exceptional method post-condition.

`invariant I`. declares a class invariant `I`. In JML, class invariants must be established by the class constructors, and must hold after any public method is called. Invariants can temporally be broken inside methods, but must be re-established before returning from them.

Figure 1 shows a fragment of the specification of a `Decimal` class. This class models floating-point numbers using two fields, `intPart` and `decPart`, of type `short`. The `intPart` field is the integer part of the floating-point number, while the `decPart` field contains the number of thousandths after the decimal point (*i.e.*, a `decPart` of `2` means `.002`, and a `decPart` of `200` means `.2`). As JML prohibits the inclusion of non-public fields in public specifications, declarations `/*@ spec_public @*/` are put immediately before `intPart` and `decPart` field declarations. These fields can thus be used in any specification. The value of a decimal number is represented by the `model` field `decimal`. A `model` field is a specification-only field used to write abstract specifications. Hence, field `decimal` models the abstract value of a decimal number, represented as `intPart * PRECISION + decPart`. Fields `intPart` and `decPart` are declared as being `in decimal`. This is useful for abstracting `assignable` specifications. For instance, `assignable decimal;` in the `normal_behavior` specification for method `setValue` says that it may modify `intPart` or `decPart` only. The class invariant restricts the values that `intPart` and `decPart` can take: `intPart` is a positive number, less than the constant `MAX_DECIMAL`, the maximal value of a short, while `decPart` ranges between `0` and the constant `PRECISION`. The value of `PRECISION` is `1000`, thus the class `Decimal` represents decimal numbers up to three decimal places. If the integer part of a decimal is `MAX_DECIMAL` then the decimal part of the decimal must be `0`.

The `normal_behavior` specification for method `setValue` says that if `v` is non-negative then the method will terminate normally, the new value for `decimal` will be `v * PRECISION`, and the value returned by the method will be the object `this`. The `exceptional_behavior` specification for method `setValue` states that if `v` is negative, the method will throw an exception of type `DecimalException` and no location will be modified. The keyword `also` expresses that `setValue` can have either a normal or an exceptional behavior.

2.2 The JML Common Tools

We tested the specifications for the Ax-LIMS project manager plugin by using the JML common tools [10]. The JML common tools is a suite of tools providing support to run-time assertion checking of JML-specified Java programs. The suite includes `jmlc`, `jmlunit` and `jmlrac`. The `jmlc` tool compiles JML-specified Java programs into a Java bytecode that includes instructions for checking JML specifications

```
public class Decimal extends Object {
 public static final short MAX_DECIMAL = (short) 32767;
 public static final short PRECISION = (short) 1000;

 /*@ spec_public @*/ short intPart = (short) 0; //@ in decimal;
 /*@ spec_public @*/ short decPart = (short) 0; //@ in decimal;

 /*@ public invariant 0 <= intPart && intPart <= MAX_DECIMAL &&
   @                  0 <= decPart && decPart < PRECISION &&
   @                  intPart == MAX_DECIMAL ==> decPart == 0;
   @ public model int decimal;
   @ public represents decimal <- intPart * PRECISION + decPart;
   @*/

 /*@ public normal_behavior
   @  requires v >= 0;
   @  assignable decimal;
   @  ensures decimal == v * PRECISION;
   @  ensures \result == this;
   @    also
   @ public exceptional_behavior
   @  requires v < 0;
   @  assignable \nothing;
   @  signals (DecimalException de) true;
   @*/
 public Decimal setValue(short v) throws DecimalException { ... }
}
```

Fig. 1. Example of a JML specification.

at run-time. The `jmlunit` tool generates JUnit [13] unit tests code from JML specifications and uses JML specifications processed by `jmlc` to determine whether the code being tested is correct. Test drivers are run by using the `jmlrac` tool, a modified version of the `java` command that refers to appropriate run-time assertion checking libraries. For the case study presented in this paper, we provided JML specifications for the main classes of the Ax-LIMS project manager plugin, however some classes regarding the Graphical User Interface (GUI) were not specified. We used `jmlunit` to generate JUnit tests for each method in every class and `jmlrac` to run the tests.

3 Outline of Ax-LIMS

Ax-LIMS is a Java Desktop project management application wrapping up in Java most services and functionalities provided by LIMS (Laboratory Information Management System), a project management application specially designed for the planning, organization and resource management of biotechnology projects at CIAT

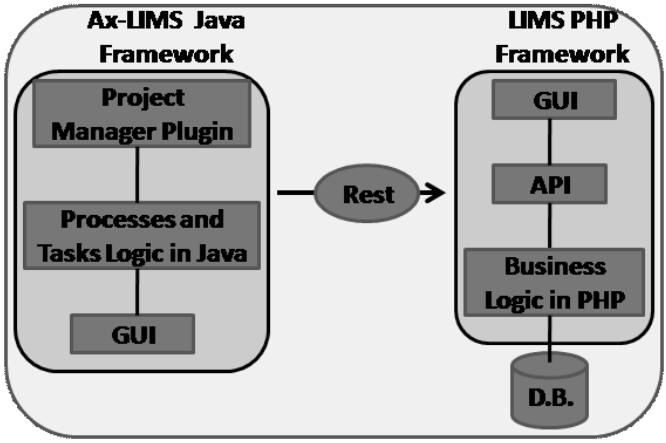

Fig. 2. Ax-LIMS Software Architecture

(International Center for Tropical Agriculture) [5]. Biotechnology projects manage laboratory processes and tasks. A process manages laboratory experiments. Tasks are administrative tasks that need to be carried out between the project start and completion dates. Information about projects is stored in a PostGreSQL database. Although LIMS business logic is written in PHP (see Figure 2), a REST (Representation State Transfer) API interface exists that communicates Ax-LIMS and LIMS. Ax-LIMS implementation relies on the project and process functionalities provided by LIMS. The initial goal of the case study included developing a project manager plugin that directly connects to the LIMS application by using the REST API interface. Developing a process manager plugin is part of a future development.

3.1 Software Requirements of the Ax-LIMS Project Manager Plugin

In an early software development stage, software requirements were gathered from discussions with the client and then written in a software requirements document. A set of use cases document describing the interaction of the final user with the application was produced. From these discussions, the Ax-LIMS project manager plugin class structure was derived. It included the classes `Project`, `Task`, `State`, `User`, `UserItem` and `Document`. The `Project` class declares a field `tasks` of type `List` for managing project tasks. The `State` class defines all possible states for projects and tasks. Classes `User` and `UserItem` implement the user interaction. Class `Document` models documents related to a particular project. The JML experts group's initial goal was turning the software requirements document into a more formal document, one that expresses requirements in terms of pre-conditions, post-conditions and invariants, suitable for formal software development and formal software checking. We summarize below these formal requirements. We intentionally omit those requirements dealing with user interaction. We then look at how these formal requirements can be expressed in a formal specification language such as JML and how run-time assertion checking with JML can be used to enforce the correct implementation of the Ax-LIMS project manager plugin.

(i) Every task has a planned start date as well as a planned completion date.

(ii) The planned start date for a task is smaller than its planned completion date.

(iii) If a task has an actual completion date, then it also has an actual start date.

(iv) The actual completion date for a task is bigger than its actual start date or it has no actual completion date.

(v) If a task has no actual start date and its planned start date is less than or equal to the system current date, then the state of the task is *uninitiated*.

(vi) If a task has no actual start date and its planned start date is greater than the system current date, then the state of the task is *initiated*.

(vii) If a task has an actual start date but does not have an actual completion date, then the state of the task is *in-progress*.

(viii) If a task has an actual completion date, then the state of the task is *finished*.

(ix) The only possible states for tasks are *uninitiated*, *initiated*, *in-progress* and *finished*.

(x) Similar project state definitions exist as for task state definitions[6]. Additionally, if a project has no actual completion date, no actual start date, no planned completion date and no planned start date, then its state is *unresolved*.

(xi) The only possible states for projects are *unresolved*, *uninitiated*, *initiated*, *in-progress* and *finished*.

(xii) If a project has an actual start date, then it also has a planned start date and a planned completion date.

(xiii) If a project has an actual completion date, then it also has an actual start date.

(xiv) The planned start (completion) date for a project is the smallest (biggest) planned start (completion) date of the tasks making up the project.

(xv) The actual start date for a project is the smallest actual start date of the tasks making up the project.

(xvi) A task refers to a single project.

(xvii) If at least one of the tasks making up a project is *in-progress*, then the project itself is *in-progress*.

(xviii) If a project is *initiated*, then none of the tasks making it up has an actual start date.

(xix) If a task is *in-progress*, then it has a planned completion date, a planned start date and an actual start date, but it does not have an actual completion date.

The first nine requirements describe task invariants. Requirements ten to thirteen model project invariants. Requirements fourteen to sixteen formalize invariants relating tasks and projects. Requirements sixteen to nineteen are requirements that are expected to hold from the invariants. These are used for checking insight on

[6] Additionally, in the definition of *uninitiated* and *initiated* for projects, the condition saying that a planned start date exists must be added in each case.

whether the specification and the actual code are correct or not.

4 Specifications for Ax-LIMS

When writing specifications, it is important to have a reasonable level of abstraction so that large changes in the implementation can involve minor changes in the specification only. As an example of how abstract the specifications for the Ax-LIMS project manager plugin are, we present the formal modeling of the project tasks. In class `Project`, a field `tasks` is declared for storing the tasks that are part of the project. Many possible implementations for field `tasks` exist. For instance, it can be implemented as an instance of the Java standard `ArrayList` class or it can be declared to be an object array of some type. However, we want that our specifications do not directly depend on the implementation. JML comes with a suite of types than can be used for defining abstract models about collection types. We use an abstract `model` variable `T` for modeling `tasks`[7] (see below). An abstract `model` variable is a specification-only variable used to write abstract specifications. Abstract specifications are related to actual Java code through the use of a `represents` clause. The type of `T` is `JMLEqualsSequence`, a JML abstract collection type. Objects in a collection of type `JMLEqualsSequence` are compared by using the method `equals` instead of comparing their references for equality. The pure method `TasksRepresentation` represents `T` as an object produced as the insertion of all the elements in the field `tasks`. Pure methods are side-effect free methods.

```
//@ public model JMLEqualsSequence T;
//@ public represents T <- TasksRepresentation();

/*@ public pure model JMLEqualsSequence TasksRepresentation() {
  @  if (this.tasks == null) return null;
  @  JMLEqualsSequence res = new JMLEqualsSequence();
  @  for (int i=0; i<tasks.size(); i++)
  @    res = res.insertBack(tasks.get(i));
  @  return res;
  @ }
  @*/
```

The definition of an abstract model variable `T` for tasks makes it possible to introduce abstract specifications. For instance, the normal postcondition for method `addTask` below ensures that after calling `addTask` with a task `t`, the number of project tasks will be equal to the number of project tasks before the method call increased by one, that the task `t` will be part of the project tasks and that the project tasks resulting from the call are obtained from the project tasks before the call after adding `t` to them. These postconditions are asserted using the abstract model variable `T`. They are therefore valid no matter which implementation for project tasks is provided. The `requires_redundantly` specification states both a method

[7] See Section 2.1 for a presentation of `model` variables.

precondition property and that it follows from other properties of the specification, *e.g.*, a class invariant stating that `T != null` exists. Redundant properties serve to bring them to the attention of the readers of the specification and the people working in the implementation of the application.

```
*@ public behavior
 @  requires_redundantly T != null;
 @  assignable T, this.psd, this.pcd, this.asd, _state;
 @  ensures T.size() == \old(T.size())+1;
 @  ensures T.has(t);
 @  ensures T.equals(\old(T).insertBack(t));
 @  ...
 @*/
public void addTask(Task t) throws ProyectoException { ... }
```

5 Ax-LIMS Plugin JML-Assisted Development

In the following, we describe how the requirements in Section 3.1 are specified in JML. Then, Section 5.1 presents some interesting aspects in the specification and development of the Ax-LIMS project manager plugin, namely, how JML can be used to recognize those pieces of Java code that need to be handled for exceptions (Exception Handling), how incorrect code can be detected (Incorrect Code) and how the iterative process of writing code, formally specifying invariant properties for it and then checking the code against the properties by using JML can be used for enforcing the writing of correct code (Enforcing Code That Fulfills Invariants).

First of all, in classes `Project` and `Task` a `ghost` variable `_state` is declared to represent all possible states for projects and tasks respectively. Like `model` variables, `ghost` variables are specification-only variables, *i.e.*, they can be mentioned within JML specifications but not in Java code. Additionally, a special `set` JML command can be used to assign the `ghost` variable a value. This is important for us as we want to set `_state` while respecting the formal requirements about states, asserted as invariants in JML. The JML assertion checker tool automatically checks whether the setting respects the invariants. A concrete variable `state` is declared to be `in _state` (see below). This is used for abstracting the project and task states from the actual code used for representing them. The `//@ public` and `/*@ spec_public @*/` JML specifications guarantee that `_state` and `state` can freely be used in JML specifications.

```
/*@ spec_public @*/ private int state; //@ in _state;
//@ public ghost int _state;
```

In classes `Task` and `Project`, fields `psd`, `pcd`, `asd` and `acd`, representing planned start date, planned completion date, actual start date and actual completion date are declared. The requirement "Every task has a planned start date as well as a planned completion date" is asserted as the following two invariants for class `Task`, which restrict fields `psd` and `pcd` to be `non_null`.

```
/*@ spec_public @*/ private /*@ non_null @*/ Date psd;
/*@ spec_public @*/ private /*@ non_null @*/ Date pcd;
```

Informal requirements such as "The planned start date for a task is smaller than its planned completion date" and "If a task has an actual completion date, then it also has an actual start date" are asserted as the two following invariants in class `Task`. The method `compareTo` in the Java standard class `Date` returns a positive number whenever the first date is bigger than the second one, a negative number when the second one is bigger than the first one, otherwise returns `0`. Notice that `pcd.compareTo(psd)` is well defined because of the `non_null` invariant specification provided for `psd` and `pcd` above.

```
//@ public invariant pcd.compareTo(psd) > 0;
//@ public invariant acd != null ==> asd != null;
```

The possible states a task can take are restricted by the first invariant below. A state requirement definition such as "If a task has an actual start date but does not have an actual completion date, then its state is *in-progress*" is asserted as a logical equivalence that restricts those cases when `_state` is allowed to take the value `State.IN_PROGRESS`. The rest of invariants for classes `Task` and `Project` are asserted in a like manner.

```
//@ public invariant _state == State.UNINITIATED ||
//@                  _state == State.INITIATED ||
//@                  _state == State.IN_PROGRESS ||
//@                  _state == State.FINISHED;

//@ public invariant acd == null && asd != null <==>
//@                  _state == State.IN_PROGRESS;
```

The specification of those requirements relating tasks and projects, *i.e.*, requirements fourteen to sixteen in Section 3.1, are of primary importance to us since they guarantee that class invariants hold through the use of other classes. As an example of the formal specification of these requirements, we show the specification of "The actual start date for a project is the smallest actual start date of the tasks making up the project" informal requirement. We formalize this requirement with the aid of the abstract variable `T` defined in Section 4. The project actual start date is `null` if and only if the actual start date for each task in `T` is `null`. Otherwise, if the project actual start date is not `null`, the existence of a smallest actual start date in `T` is restricted to be equal to the project start date. In the code below, `T.get(i).getAsd()` takes the `i`-th task in `T` and returns its actual start date.

```
/*@ public invariant asd == null <==>
  @  (\forall int i; i>=0 && i<T.length(); T.get(i).getAsd()==null);
  @
  @ public invariant asd != null ==>
  @  (\exists int i; i>= 0 && i<T.length();
```

```
  @    T.get(i).getAsd() != null &&
  @    asd.compareTo(T.get(i).getAsd()) == 0);
  @
  @ public invariant asd != null ==>
  @  (\forall int i; i>= 0 && i<T.length();
  @    T.get(i).getAsd() == null ||
  @    asd.compareTo(T.get(i).getAsd()) <= 0);
  @*/
```

5.1 Interesting Aspects of Specification and Implementation

Exception Handling.

JML can be used to detect those pieces of code that need to be handled for exceptions. As an example of this, the original code developed by the ParqueSoft engineers for setting a project identifier is shown below.

```
public void setProjectId(Long id) { this.projectid = id; }
```

In addition to the requirements in Section 3.1, in order to keep consistency with the database, project identifiers in class `Project` are required to be non-null. This requirement is asserted as the invariant `/*@ non_null @*/ Long projectid;` in class `Project`, which disallows `projectid` from being `null`. When checking the method `setProjectId`, JML reports an error indicating that the method breaks the asserted invariant. We thus need to modify the code for the method so as to comply with the invariant: we add Java code that throws an exception whenever `id` is `null`. The modified code is shown below. The incorrect assignment of `null` to `projectid` could have been found by careful inspection of code, but writing a formal specification forces one to think of the problem, and using a tool for checking the specifications ensures that most cases in most places in the code are considered.

```
public void setProjectId(Long id) throws ProjectException {
 if (id == null) throw new ProjectException();
 this.projectid = id;
}
```

Incorrect Code.

We show how JML was used to the detect the incorrect code of method `setTasks` in class `Project` (see below). Method `setTasks` takes a list of tasks and assigns it to the field `this.tasks`. The implementation of `setTasks` uses bad programming practices. First, a reference to an object of type `List` is locally assigned rather than the elements of the list added to the field `this.tasks`. Adding the elements one by one additionally guarantees that the project state is re-established each time. Second, the elements of `this.tasks` are not eliminated before the elements of the list are added to `this.tasks`. Note that assigning `null` to the field `this.tasks` so

as to eliminate its elements would have also been bad programming practice, as the project would probably be in an inconsistent state afterwards.

```
public void setTasks(List t) { this.tasks = t; }
```

Thinking about invariants prior to writing code is a practice to which programmers do not easily adhere. Having a previous formal specification of the application and systematically using a tool for checking the correctness of the code as it is written forces programmers to think about how the written code affects the consistency and the correctness of the whole program. We present below a correct implementation for the method `setTasks`. The implementation relies on method `delTask` for removing the current project tasks and `addTask` for adding the new tasks. The correctness of the implementation of `setTasks` depends on the correctness of the implementation of both `delTask` and `addTask`, *e.g.*, for re-establishing the state of the project each time a task is deleted or added.

```
public void setTasks(List tasks) {
 int size = this.tasks.size()-1;
 for(int i=size; i>=0; i--) delTask(this.tasks.get(i));
 for(int i=0; i<tasks.size(); i++) addTask(tasks.get(i));
}
```

We can further use JML for checking insight on how method `setAsd` works in some particular cases. To do this, we use the JML `assert` specification construct, which checks for satisfaction of a predicate at a dedicated point within a method. For instance, we are interested in knowing what the state of a project would be if `setAsd` is called with an empty list of tasks. We assert the specification below immediately before the end of method `setTasks`. JML reports errors for any value of `_state` other than `State.UNRESOLVED`.

```
//@ assert tasks.size() == 0 ==> _state == State.UNRESOLVED;
```

Enforcing Code That Fulfills Invariants.

The iterative process of writing code, formally specifying invariant properties for it and checking the code against the properties by using JML can be used for enforcing the writing of correct code. As an example of this, the original code written by the ParqueSoft engineers for setting the actual starting date for a project is shown below (see method `setAsd`).

```
public void setAsd(Date asd) { this.asd = asd; }
```

When checking the correctness of the implementation of method `setAsd`, JML reports some errors indicating that `setAsd` breaks the invariant below as well as any invariant involving `asd` and `_state`.

```
//@ public invariant acd == null && asd != null <==>
//@                  _state == State.IN_PROGRESS
```

Therefore, besides assigning a value to `asd`, method `setAsd` must also set the project state accordingly. After several iterations of specifying and checking, we came up with an appropriate JML code for setting the project state. The JML specification code is shown below together with the actual Java code. The Java predicate expression `c ? s : t` returns `s` if the evaluation of `c` is true, otherwise it returns `t`. We use the `set` command of JML for setting the `ghost` variable `_state`.

```
public void setAsd(Date asd) {
 this.asd = asd;

 /*@ set _state =
   @  this.acd != null ? State.FINISHED :
   @   this.asd != null ? State.IN_PROGRESS :
   @    this.psd == null ? State.UNRESOLVED :
   @     this.psd.after(Calendar.getTime()) ? State.UNINITIATED :
   @                                          State.INITIATED;
   @*/
}
```

When checking the new code for `setAsd`, JML reports some errors indicating that the implementation of `setAsd` breaks the three invariants below. All three invariants refer to field `asd`.

```
/*@ public invariant asd != null ==> psd != null && pcd != null;
  @ public invariant acd != null ==> asd != null;
  @ public invariant acd != null ==> acd.compareTo(asd) > 0;
  @*/
```

We thus need to modify the implementation of method `setAsd` once again. We should add code that raises an exception when any of the invariants are broken (see below). In addition to this code, code that complies with the invariant "The actual start date for a project is the smallest actual start date of the tasks making up the project" must also be added to `setAsd`. In the interest of brevity we do not show the JML code dealing with this invariant here.

```
public void setAsd(Date asd) throws ProjectException {
 if((acd != null && (asd == null || acd.compareTo(asd) <= 0)) ||
    (asd != null && this.psd == null))
   throw new ProjectException();

 this.asd = asd;

 /*@ set _state =
   @   this.acd != null ? State.FINISHED :
   @    this.asd != null ? State.IN_PROGRESS :
   @     this.psd == null ? State.UNRESOLVED :
```

```
   @      this.psd.after(Calendar.getTime()) ? State.UNINITIATED :
   @                                           State.INITIATED;
   @*/
}
```

6 Related and Future Work

In [3] N. Cataño and M. Huisman report on a case study in which a freely available electronic purse application is specified and checked using ESC/Java. The process of checking revealed several errors in the implementation of the purse. The work presented here is situated on a different axis. We do not check a released application, but use formal specifications and a formal methods tool to accompany the development of the application. We use the insight provided by the tool to modify the implementation accordingly or evolve the specifications.

J.-L. Lanet *et al.* report on a test case study applied to a JML-specified Java bank application. The main goal of their work is to find inconsistencies between the JML specifications and the code, and between these and the informal requirements of the application. Unit tests are automatically generated by using a customized tool. They investigated whether JML specifications written for proof are well suited for validation by test. In the case study described here JML is directly used for checking the JML specifications.

Executable specifications provide a way of checking if a formal specification is flawed, *i.e.*, that no implementation satisfying the specification exists, and when the formal verification of the final software system is envisioned, executable specifications help not to add extra-weight to the verification process by ruling out all those behaviors for which no implementation exists. The work presented here can be regarded as part of a software engineering methodology that envisions the execution of specifications before the checking for correctness is performed. We plan to use the jmle tool [9] to execute the JML specifications written for Ax-LIMS so as to check their consistency. The jmle tool executes JML specifications by translating them to constraint programs. We believe that applying different tools at different stages of software development makes it possible to find different kind of specification and implementation errors, and additionally to alleviate the work carried out by other tools. We would additionally like to compare standard software design coupled with JML against using B [1] from scratch as a specification, modeling and implementation methodology.

7 Conclusion

We have presented a case study in formal software development of a Java Desktop project management application using JML. For the case study, we formed a group of software engineers and JML experts for developing a project manager plugin application. Our goals for the case study included determining how the use of JML for modeling software requirements can enforce the programming of correct Java

code, how a group of JML formal methods experts can effectively contribute to the making of decisions within a software development team and how JML can be used to corroborate or refute the software development team's insights on how an application should work.

For the case study, considerable time was spent in discussing about the correct understanding of the informal requirements of the Ax-LIMS project manager plugin. Once these informal requirements were clear, writing JML specifications for them was straightforward. JML was found to be expressive enough to formalize the Ax-LIMS plugin informal software requirements. Most of these requirements precisely describe the states for projects and tasks managed by Ax-LIMS.

The initial versions of the code written by the ParqueSoft engineers contained about 40 programming errors that the JML experts group reported to them. This allowed the engineers to write a correct version of their code. We employed the `jmlunit` tool to generate JUnit tests for each method in every class. In total, 6322 unit tests were generated for class `Task`, 5355 for class `Project`, 866 for classes `User` and `UserItem`, and 695 for class `Document`. The reported errors were not very intricate in general. They could have been detected by careful code-inspection, but writing a formal specification expressing the software requirements of the application forced the engineers to do code-inspection. Additionally, JML guarantees that most cases are considered, without having to put much effort in writing test scenarios since they are automatically generated from the JML specifications.

Several factors contribute to why the software industry is reluctant to use formal methods as part of its software engineering practices. First of all, people in industry are unwilling to adopt formal methods because of the mathematical aspects underlying formal methods tools and techniques, which can be difficult for non-experts to assimilate. However, JML specifications are designed to be easy and accessible for an average Java programmer as they use a Java-like syntax. Second, people in industry have the perception that the use of formal methods slows down projects. With the growing tendency to deliver products as fast as possible so as to meet deadlines, little investment in software reliability is made in general. For this case study, we believe that we accelerated the development of the Ax-LIMS project manager plugin by pointing out many errors at an early stage. This is important as the earlier an error is discovered the less serious the consequences it can inflict on the final software system.

In previous software developments carried out by the ParqueSoft engineers, OCL (Object Constraint Language) was used for expressing constraints on the UML models. However, for this case study, the ParqueSoft engineers decided to use JML for expressing these constraints instead. The ParqueSoft engineers found JML specifications much easier to code than OCL constraints. JML will therefore be used as the language for expressing model constraint in future ParqueSoft Java-based software developments.

References

[1] J.R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 2008.

[2] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, June 2005.

[3] N. Cataño and M. Huisman. Formal specification of Gemplus' electronic purse case study. In Lars-Henrik Eriksson and Peter A. Lindsay, editors, *FME: Formal Methods Europe*, volume 2391 of *Lecture Notes in Computer Science*, pages 272–289, Copenhagen, Denmark, July 22-24 2002. Springer.

[4] N. Cataño and M. Huisman. Chase: A static checker for jml's assignable clause. In *VMCAI 2003: Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 26–40, London, UK, 2003. Springer-Verlag.

[5] International Center for Tropical Agriculture, 2008. `http://www.ciat.cgiar.org/`.

[6] The ESC/Java 2 Tool. `http://secure.ucd.ie/products/opensource/ESCJava2/`, 2008.

[7] The JACK tool. `http://www-sop.inria.fr/everest/soft/Jack/jack.html`, 2008.

[8] The Krakatoa tool. `http://krakatoa.lri.fr/`, 2008.

[9] Ben Krause and Tim Wahls. jmle: A tool for executing JML specifications via constraint programming. In L. Brim, editor, *Formal Methods for Industrial Critical Systems (FMICS '06)*, volume 4346 of *Lecture Notes in Computer Science*, pages 293 – 296. Springer-Verlag, August 2006.

[10] Gary T. Leavens. The Java Modeling Language (JML) home page, 2008. http://www.jmlspecs.org.

[11] Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion, Minneapolis, Minnesota*, pages 105–106. ACM, October 2000.

[12] G.T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, and P. Chalin. JML reference manual. `http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman_toc.html`, 2008.

[13] Johannes Link. *Unit Testing in Java*. Morgan Kaufmann, 2003.

[14] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, Oct. 1992.

[15] ParqueSoft, 2008. `http://www.parquesoft.com`.

[16] Unified Modeling Language, 2008. `http://www.php.net/`.

[17] J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In *Proceedings of TACAS*, number 2031 in LNCS, pages 299–312. Springer, 2001.