JFly: A JML-Based Strategy for Incorporating Formal Specifications into the Software Development Process

Nestor Catano[1], João Pestana[2], and Ricardo Rodrigues[2]

[1] The University of Madeira, M-ITI, CMU-Portugal
ncatano@m-iti.org
[2] The University of Madeira, Portugal
{joao.pestana,ricardo.rodrigues}@max.uma.pt

Abstract. This paper presents JFly, a JML-based strategy for incorporating formal specifications into the software development of object oriented programs. The strategy consists in evolving functional requirements into a semi-formal requirements form, and then expressing these requirements as JML formal specifications. What makes our strategy different from existing strategies is the particular use of JML we make all along the way from requirements to validation-and-verification. We validate our strategy with the formal development of a smart card application for managing medical information.

1 Introduction

Although software engineering methods provide a disciplined approach to software development, it is still quite common to find flawed software systems. A way to construct correct programs is through the use of formal specifications as part of a software engineering practice. In this paper, we propose JFly, a strategy that incorporates formal specifications into the software development process of object oriented programs by evolving informal functional requirements into formal specifications (Section 2). Having informal functional requirements modelled in a formal specification language allows for the use of formal methods tools for checking program correctness. We use JML [11] as the formal specification language for writing our specifications. Our strategy consists in evolving informal functional software requirements written in natural language (e.g. English or Portuguese) into semi-formal requirements, i.e. requirements written in natural language, yet in a more mathematical style. Hence, informal requirements are written as semi-formal functional requirements of the form if <event> then <restriction>, or as semi-formal class and system invariants. Semi-formal requirements are then written as JML method specifications and JML class invariants respectively. We validate our strategy with the development of a HealthCard smart card application (Section 3). We used the JML Common tools [3] to check the HealthCard for correctness. Neither using formal specifications to increase confidence in a system implementation nor gradually transforming software requirements from a high level of abstraction to a more concrete level are new

INForum 2010 - II Simpósio de Informática, Luís S. Barbosa, Miguel P. Correia (eds), 9-10 Setembro, 2010, pp. 19–30

ideas [6, 8, 10]. What makes our strategy different from others is the particular use of JML specifications we make all along the way from requirements to validation-and-verification.

L. Shaoying et al. in [15] propose the SOFL methodology for software development. The SOFL language integrates Data Flow Diagrams, Petri Nets, and VDM-SL. Likewise JFly, in SOFL, the process of writing formal specifications is the result of a 3-step process that evolves informal specifications into an abstract formal specification. The SOFL methodology has established a much more mature technique for developing formal design specifications for software systems than the JFly strategy presented by us. Nonetheless, having formal specifications expressed directly in JML, rather than using an abstract general form, allows the direct use of JML tools, which implement various formal checking techniques. JML is a formal specification language that uses a syntax close to Java syntax and hence Java programmers find JML easy to use, which helps to bridge the gap between mathematical formalisms and software engineering techniques [4].

Supplementary to our work, V. T. Vasconcelos et al. have implemented the ConGu tool [16], which reduces the problem of checking algebraic specifications to the run-time monitoring of contracts described in JML. Their work extend our work in a way that further considers algebraic specifications to express correct software components.

Finally, M. G. Ilieva and O. Ormandjieva have studied the automatic translation of software requirements written in natural language into formal specifications [9]. Our work is less ambitious, yet more practical.

1.1 The Java Modeling Language (JML)

JML is a behavioral interface specification language for Java, which means that the only correct implementation of a JML class specification is a Java class implementation with the specified behavior. JML is now an academic community effort with many groups developing tools to support JML [3]. In JML, methods are specified using `requires`, `modifies`, and `ensures` clauses, which give the precondition, the frame (what locations may change from the pre- to the poststate), and the postcondition respectively. A method specification can also include an `exsures` or `signals` clause to specify conditions under which the method could throw an exception. Class invariants can also be written to constrain the states of objects. JML specifications use Java syntax and are embedded in Java code between special 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.

1.2 The JML Common Tools

The JML common tools [3, 2] is a suite of tools providing support to run-time assertion checking of JML-specified Java programs. The suite includes `jml`, `jmlc`, `jmlunit` and `jmlrac`. The `jml` tool checks the JML specifications for syntax errors. The `jmlc` tool compiles JML-specified Java programs into a Java bytecode that includes instructions for checking JML specifications at run-time. The `jmlunit` tool generates JUnit [12] unit tests code from JML specifications and uses JML specifications processed by `jmlc` to determine whether the code being tested is correct or not. 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.

The JML common tools make possible the automation of regression testing from the precise, and correct JML characterisation of a software system. The quality and the coverage of the testing carried out by JML depend on the quality of the JML specifications. The run-time assertion checking with JML is sound, *i.e.*, no false reports are generated. The checking is however incomplete cause users can write informal descriptions in JML specifications, e.g., `(* x is positive *)`. The completeness of the checking performed by JML depends on the quality of the specifications and the test data provided.

2 JFly: the Proposed Strategy

We have developed a strategy for evolving informal functional requirements into formal specifications, which can be employed as part of existing object-oriented software development methodologies [14] (Chapter 28). Hence, software developers must define precise interface specifications for underlying software components, based on data types and the conceptual metaphor of the design-by-contract [13]. The strategy consists in incorporating informal, semi-formal, and formal specifications all along an existing object-oriented software engineering methodology. In Figure 1, we do not restrict any phase to occur before or after any other phase, so that arrows convey information on usage rather than on precedence in time. Software development phases are iterative so that they can be revisited at later phases to obtain a correct implementation of the system. During the analysis phase, "informal" functional requirements are written (functional requirements written in natural language). As informal functional requirements are expressed in a natural language, inconsistencies can be introduced during the analysis phase. Hence, informal functional requirements are

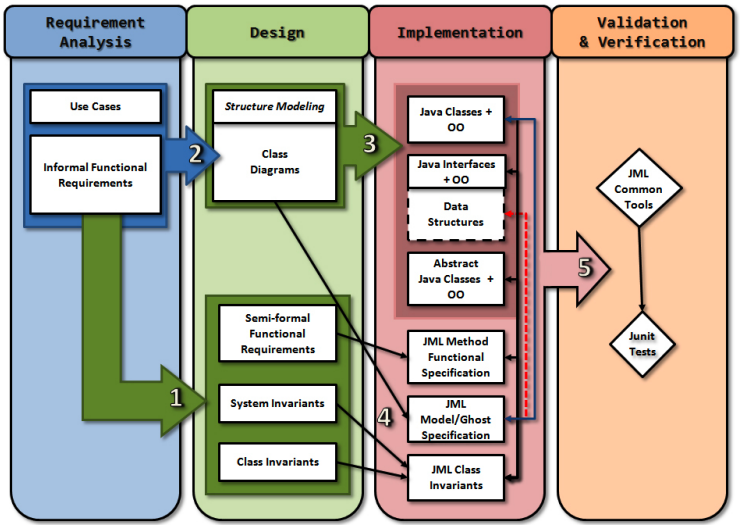

Fig. 1. The Software Development Process

first evolved into "semi-formal" requirements (see Arrow 1), and then ported into JML formal specifications (see Arrow 4). Having formal specifications expressed in JML makes it possible to use JML-based formal methods tools to check for flaws. The semi-formal requirements are divided into three parts, namely the semi-formal functional requirements (ported into JML method specifications), the class invariants, and the system invariants (these two are ported into JML class invariant specifications). Evolving the informal functional requirements into semi-formal ones involves expressing informal requirements into an if <event> then <restriction> form (see Section 3.3 for details).

During the design phase, the requirements gathered from the analysis phase are used to define the structure of the system (see Arrow 2), which is later used to write classes, their attributes, their methods, and the relations among them (see Arrow 3). These classes are specified with JML as described above. During the implementation, we start by writing Java interfaces and Java abstract classes. From the semi-formal requirements, JML functional specifications are written for abstract methods in Java interfaces and abstract classes, and JML class invariants are written, modelling global properties of the system. Finally, JML abstract variables are defined to describe the distinct abstract data types used in the application, and how they are manipulated through class inheritance (see Section 3.5). JML specifications provide support to the writing of correct code for concrete classes that implement the interfaces and the abstract Java classes. JML specifications also provide support to a business contract programming style of programming, in accordance with Bertrand Meyer's design-by-contract principles.

During the validation-and-verification phase, the implementation is checked against the JML specifications (Arrow 5), using the JML Common Tools [3]. If they issue an error, then the implementation or the specifications are evolved accordingly. Therefore, it is possible to go back to a previous phase and make amendments to the JML specifications or the implementation itself. Notice that inconsistencies can be detected before an implementation for the system is written. For instance Java interfaces and Java abstract classes are checked against JML specifications before writing an implementation for these classes and interfaces, or JML specifications can be validated in isolation [5].

3 A Running Example

3.1 The HealthCard Application

In the following, we describe the HealthCard smart card application we used to validate our strategy. HealthCard stores people's medical information. Smart cards are pocket-sized plastic cards with embedded integrated circuits that process data. A typical smart card application includes on-card applets (the applets running on the card), a card reader-side, and off-card applications (e.g. a computer program communicating with the card applets). HealthCard is written in Java Card, a subset of Java used to program card applets. We used the Java Card Remote Method Invocation (JCRMI) model for communication between off-card applications and on-card applets. This model implements a client-server setting with the HealthCard acting as server, and off-card applications as clients, communicating via APDU (Application Protocol Data Unit) messages. Figure 2 shows the structure of the HealthCard. A patient can use his HealthCard to furnish accurate medical information to general practitioners in medical centres with the appropriate system to read it. The HealthCard manages the patient's personal details, his allergies, his historical record of vaccines, diagnosis, treatments and prescribed medicines. The HealthCard is divided in several modules for managing medical information. Each module has a remote interface and an implementation class that serves the appropriate services. All the remote interfaces are referenced in a single remote interface named `CardServices` whereby an external client can invoke services. Hence, if an external client calls the method `getApp()` in `CardServices`, it gets a reference to the `Appointments` remote interface. This reference is then used to invoke appropriate methods implementing services.

3.2 Informal Functional Requirements

Informal functional requirements define, in an informal way, the inputs, the behavior, and, in general, functional restrictions of the system to develop. In the following, we present a small example from the HealthCard system that shows how informal functional requirements are evolved into the three kind of semi-formal requirements described in Section 2. We present below some of the informal requirements of the HealthCard application.

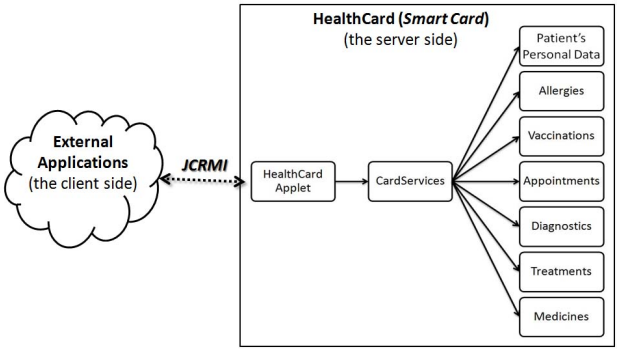

Fig. 2. The Health Card System Structure

IFR1 There must not exist duplicated entries for allergies with the same designation code.
IFR2 A fixed number of allergies can be introduced in the card only.
IFR3 All allergy designation codes must have a stipulated length.
IFR4 The prescription date of a medicine must be bigger than or equal to the date of the appointment in which the medicine was prescribed.

The following sub-sections show how the informal requirements above are evolved into semi-formal requirements. Evolving informal requirements into semi-formal requirements is not a deterministic process, nonetheless it obeys general guidelines that proved to be practical in the development of the HealthCard. Informal requirements involving several domain concepts (e.g. medicine and appointment in IFR4) are evolved into system invariants; informal requirements involving or restricting a single domain concept (e.g. allergy in IFR3) are evolved into class invariants; informal requirements functionally restricting a single domain concepts under certain events (e.g. IFR1 and IFR2) are evolved into semi-functional requirements that follow an if <event> then <restriction> mathematical form, which is close to a JML method specification style.

Because of the ambiguous essence of natural language, our guidelines to transform informal requirements into semi-formal ones is neither sound nor complete, nonetheless our experience shows that is useful in practice. We have implemented a prototype script shell based tool doing the transformation automatically (see Section 3.8).

3.3 Semi-Formal Functional Requirements

The informal functional requirement IFR1 is transformed into if <a new allergy is to be added to the list of referenced allergies and the allergy designation has already been referenced>, then <the new allergy is not inserted>. We show below the semi-formal requirements obtained from the first two informal requirements above.

SFR1 From IFR1. If a new allergy is to be added to the list of referenced allergies and the allergy designation has already been referenced, then the new allergy is not inserted.

SFR2 From IFR2. If an allergy is to be added to the list of referenced allergies and the limit of the number of referenced allergies has already been attained, then the state of the card remains unchanged.

3.4 Class and System Invariants

Class invariants are written from informal requirements that describe limitations or constraints on a small-scale, e.g. limitations of properties that eventually will restrict or describe a certain domain concept only. Hence, the informal functional requirement IFR3: "All allergy designation codes must have a stipulated length", restricts the length of the designation code of any "allergy". To write this class invariant (see CI1 below), we use a variable des to represent the designation code of the allergy. This variable can be modelled as a JML `abstract` variable (see Section 3.5). CI1 describes a property about the length of the code of an allergy, so that eventually it will become a class and the code a `static` field of it.

CI1 invariant size(des) == CODE_LENGTH

Unlike class invariants, system invariants describe invariant properties relating several domain concepts. For instance, IFR4 describes a property on medicines, managing information on prescribed medicines in appointments, and appointments, managing information about appointments scheduling. IFR4 becomes the semi-formal system invariant requirement SI1 below.

SI1 For all object m of type medicine, and all object a of type appointment such that appointment(m) is equals to a, then date(m) is bigger than or equal to date(a).

3.5 Design and Implementation

During the design phase, the structure of the application is created from the requirements. This structure encompasses class diagrams for interfaces, abstract classes, and concrete classes. In parallel to this phase, semi-formal functional requirements and class and system invariants are written (Sections 3.3 and 3.4). Semi-formal specifications are later ported to JML specifications (Section 3.6). During the implementation phase, from the structure of the application generated in the design phase, Java abstract classes, Java interfaces, and Java classes are written. In a first stage, the implementation only contains code skeletons, so no method in any concrete class is implemented. JML specifications are embedded within the code. Hence, the JML Common Tools can be used to check the code during early stages of the implementation (i.e. before fully implementing concrete Java classes). Therefore, the Java code can be evolved so as to conform to the JML specifications, or the specifications can be evolved so as

to conform to an expected behavior. Checking that one conforms to another is done automatically with the JML Common Tools. JML eliminates programmers' responsibility of keeping track of how properties a program must respect are affected by changes in the code.

Furthermore, to have a high level of abstraction in specifications, JML provides support to the use of abstract variables, which exist at the level of the specification, but not in the implementation. Declarations of abstract variables are preceded by the JML keyword `model` and are related to Java code by a `represents` clause[3]. This clause specifies how the value of an abstract variable is calculated from the values of concrete variables (see Section 3.6). Abstract variables are useful in describing properties about interfaces because these are not allowed to declare (concrete) variables in Java. Within an interface, an abstract variable describes the state of the implementing classes. Abstract variable specifications for interfaces and for abstract classes do not need to be written down again in implementing classes or sub-classes, since JML specifications are inherited. This ensures behavioral sub-typing through which a sub-class object can always be used where a super-class object is expected.

3.6 JML Formal Specification

Semi-formal functional requirements SFR1 and SFR2 relate to method `addAllergy` in interface `Allergies` (see below). In Java, interfaces cannot declare attributes, hence `Allergies` declares an `abstract` JML variable `as`, modelling stored referenced allergies. The JML `JMLEqualsSequence` type models a sequence of objects that can be compared using the standard method `equals`. We declare two additional abstract variables, `size` and `maxsize`, modelling the number of referenced allergies and the maximum number of referenced allergies. A normal behavior specification expresses that if all the pre-conditions hold (clauses `requires`) in the pre-state of the method, it will terminate in a state in which all the post-conditions (clauses `ensures`) hold. SFR2 is expressed as the JML pre-condition `size < maxsize`. SFR1 appears in two separated normal postconditions that make use of the abstract method `existsAllergy` (not shown here) for checking whether the `designation` of an allergy has already been stored in `as` or not. Therefore, if the designation has already been stored, the list of allergies remains unchanged, `as.equals(\old(as))`, otherwise the allergy designation is stored at the end of the list, `as.equals(\old(as).insertBack(desigRepr(-designation)))`. JML abstract method `desigRepr` (not shown here) maps an array of bytes to a unique value. JML only allows side-effects free methods within specifications. This is enforced by using the JML keyword `pure`. All the methods used within specifications in our examples are `pure`, e.g. `existsAllergy` and `insertBack` (which uses an auxiliary `clone()` method) in `addAllergy`.

```
//@ model instance JMLEqualsSequence as;
//@ model instance short size;
```

[3] JML also provides `ghost`, a more limited variation of `model` variables.

```
//@ model instance short maxsize;

/*@ public normal_behavior
  @  requires size < maxsize;
  @  requires designation != null && date != null;
  @  requires existsAllergy(designation);
  @  ensures as.equals(\old(as));
  @    also
  @ public normal_behavior
  @  requires size < maxsize;
  @  requires designation != null && date != null;
  @  requires !existsAllergy(designation);
  @  ensures as.equals(\old(as).insertBack(
  @                       desigRepr(designation)));
  @*/
public abstract void addAllergy ( byte[] designation,
                          byte[] date)
              throws RemoteException, UserException;
```

Abstract specifications are related to actual Java code through the use of the JML `represents` clause. Hence, `as`, declared in `Allergies`, is related to code in the `Allergies_Imp`, which implements `Allergies`. The abstract variable `size` is represented as the concrete field `nextFree`, and `maxsize` as the static variable `MAX_ITEMS`. The pure method `allergiesRepr` represents `as` as a `JMLEquals-Sequence` produced by the insertion of all the elements in `allergies`. In JML, `pure` methods are side-effect free methods.

```
//@ represents size <- nextFree;
//@ represents maxsize <- MAX_ITEMS;

//@ represents as <- allergiesRepr();

/*@ pure model JMLEqualsSequence allergiesRepr() {
  @  JMLEqualsSequence r = new JMLEqualsSequence();
  @    for (short i=0; i < nextFree; i++) {
  @      r = r.insertBack((Object)(allergies[i]));
  @    }
  @  return r;
  @ }
  @*/
```

JML Class and System Invariants The Class invariant CI1 is expressed as the JML invariant below. This invariant is declared in class `Allergy`.

```
//@ instance invariant des.size == CODE_LENGTH;
```

The System invariant SI1 is expressed as the JML invariant below. This invariant suggests that a global access to medicines and appointments in the card must exist. Following the Java Card Remote Method invocation (JCRMI)

approach for communication, in which the Java Card applet is the server, the HealthCard application defines an interface `CardServices` that declares all the services available for remote objects. Class `CardServices_Imp`, an implementation of this interface in Java, accesses the information and the state of any remote object in the card. `CardServices_Imp` declares two variables `med` and `app` for keeping track of medicines and medical appointments respectively. Method `getData()` returns an array of objects of type `Medicine`. Method `getApp()` returns an array of objects of type `Appointment`.

```
/*@ invariant
  @ (\forall int i; i<med.getData().length & i>=0;
  @  (\forall int k; k<app.getApp().length & k>=0;
  @     med.getData()[i].getAppID() ==
  @                    app.getApp()[k].getID()
  @       ==>
  @     med.getData()[i].getDate() >=
  @                    app.getApp()[k].getDate() ))
  @*/
```

3.7 Validation and Verification

We used the JML Common Tools suite [3] to check our implementation of the HealthCard. This suite provides support to the run-time assertion checking of JML specifications. Checking an application with this suite is an iterative process of checking the implementation with respect to the JML specifications, and then evolving either the specification or the implementation (or both) when a run-time error is produced. Errors can be detected before a concrete implementation for the application is written. For instance, Java interfaces and Java abstract classes are checked against JML specifications before writing full implementations for those interfaces and abstract classes. At this point, programmers can go back to an earlier development phase, e.g. modifying some informal functional requirements; thereafter JML specifications are evolved accordingly.

3.8 A JFly Prototype Tool

We have built a prototype tool that automates the process of writing JML formal specifications from simple semi-formal specifications. The prototype tool builds on the idea that semi-formal specifications can be written as requirements of the form if <event> then <restriction>. Therefore, after if and before end, a method precondition exists; and after then a method postcondition occurs. The tool can be reached at `http://www.knowmydream.com/Projects/jfly/`. This is just a prototype tool; it demonstrates how our ideas on JML-based strategy for incorporating formal specifications to the software development of programs can be automated. For instance, the prototype tool transforms the specification if <date is NOT EQUAL TO null AND date's length is EQUAL TO date_model's length> then <date is EQUAL TO date_model>.

```
/*@ public normal_behavior
  @  requires date != null &&
  @           date.length == date_model.length;
  @   ensures date == date_model;
  @*/
```

4 Conclusion

We propose a strategy for evolving informal requirements into formal specifications as part of a software engineering methodology. However, evolving informal requirements into formal specifications is not a linear process: it requires great ingenuity and experience. Although we presented our ideas through the development of a smart card application, we consider that our strategy is suitable for developing correct applications that implement a client-server architecture with a need of a light-weight server specification in general. In this client-server setting, validations of methods' pre-conditions are not carried out within methods' implementations. It is the client's responsibility to ensure that methods are called with the right parameters. This reduces the size of implemented methods. This is particularly important for smart cards whose generated byte-code cannot be bigger than a certain limit to be installed on the smart card. We used JML tool machinery to check that the methods are always called with the right parameters throughout the whole application. This prevents programmers from making validations both inside and outside methods, a common programming mistake. Yet, we chose JML as the formal specification language, our ideas can also be adapted to the development of C++ programs, with formal specifications written in the ACSL (ANSI/ISO C Specification Language) [1] language instead, and the verification work accomplished with the Frama-C Tool [7].

We want to emphasise the importance of thinking of invariant properties when developing software. Thinking about invariants prior to writing code is a practice to which programmers do not easily adhere. Having a formal specification of an application and systematically using a tool, i.e. the JML Common 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 program. It is our experience that invariants are the key notion in formal software development that makes a difference with respect to traditional (non formal methods based) software engineering methodologies [4]. In general, programmers feel intimidated by the idea of coming up with an invariant. Often, they design code that can make their programs be in an inconsistent state. We strongly believe JML helps in this sense, from furnishing a friendly Java-like syntax, to making it possible to use first-order logic predicates in JML specifications naturally.

Finally, the HealthCard application consists of 20 interfaces, 16 concrete classes, and 174 KB in total, with about 4300 lines of code, of which 50% are specifications, 42% are code, and 8% include both specifications and code. The whole development can be reached at `https://sourceforge.net/projects/-healthcard/`. It took about 4 months time to the second and third authors to

write the HealthCard, supervised by the first author, who has a large experience with JML.

References

1. P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. ACSL: ANSI/ISO C specification language. `http://frama-c.cea.fr/download/-plug-in_development_guide.pdf`.
2. C. Breunesse, N. Catano, M. Huisman, and B. Jacobs. Formal methods for smart cards: An experience report. *Science of Computer Programming*, 55(1-3):53–80, March 2005.
3. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, June 2005.
4. N. Catano, F. Barraza, D.García, P. Ortega, and C. Rueda. A case study in JML-assisted software development. In P. Machado, A. Andrade, and A. Duran, editors, *Brazilian Symposium on Formal Methods (SBMF)*, pages 5–21, August 2008.
5. N. Catano and T. Wahls. Executing JML specifications of java card applications: A case study. In *24th ACM Symposium on Applied Computing, Software Engineering Trac (SAC)*, Waikiki Beach, Honolulu, Hawaii, March 8-12 2009.
6. E. W. Dijkstra. *A Discipline of Programming*. Series in Automatic Computation. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1976.
7. The Frama-C Tool. `http://frama-c.cea.fr`.
8. M. Fraser, K. Kumar, and V. K. Vaishnavi. Strategies for incorporating formal specifications in software development. *Communincations of ACM*, 37(10):74–86, 1994.
9. M. G. Ilieva and O. Ormandjieva. Automatic transition of natural language software requirements specification into formal presentation. In *Applications of Natural Language to Information Systems (NLDB)*, pages 392–397, 2005.
10. R. A. Kemmerer. Integrating formal methods into the development process. *IEEE Software*, 7(5):37–50, 1990.
11. G. Leavens, A. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.
12. J. Link. *Unit Testing in Java*. Morgan Kaufmann, 2003.
13. B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.
14. B. Meyer. *Object Oriented Software Construction*. Prentice Hall PTR, 1997.
15. L. Shaoying, A. Offutt, C. Ho-Stuart, Y. Sun, and M. Ohba. SOFL: a formal engineering methodology for industrial applications. *IEE Transactions on Software Engineering*, 24, 1998.
16. V. T. Vasconcelos, I. Nunes, and A. Lopes. Monitoring java code using ConGu. In *19th International Workshop on Algebraic Development Techniques (WADT)*. Universit di Pisa, 2008.