Research Proposal NWO Open Competition 2004
"SPEC-DEV"
Theory and Tools to Support JML Specification Development

1a) Project Title: Theory and Tools to Support JML Specification Development
1b) Project Acronym: SPEC-DEV
1c) Principal Investigator: Dr. Erik Poll

2a) Summary

The Java Modeling Language (JML, `www.jmlspecs.org`) is a specification language aimed at specifying the detailed behavior of Java programs. Over the past few years, JML has become the *de facto* standard formal specification language for Java in the academic community and in the smart card industry. A large number of tools now support JML [3], covering a wide spectrum of tool support, ranging from basic parsing and typechecking, to runtime checking, and to more advanced techniques for static checking and full-blown program verification.

The most important challenge in using these tools is the development of JML specifications themselves. JML specifications can be long and verbose, given that JML is a relatively low level specification language, and the development and maintenance of larger collections of JML specifications is a serious challenge. Moreover, JML does not provide any design methodology for developing specifications.

This research proposal addresses this challenge, by developing a tool-supported high level specification language, called JBON. JBON is a higher-level specification language than JML, more along the lines of UML and, more specifically, (E)BON. JBON is tailored to JML in the same way that JML is tailored to Java. We believe that JBON can provide a vital next step for the growing impact of JML and the family of tools that support JML.

2b) Abstract for laymen (in Dutch)

Het ontwikkelen van talen en methoden voor het specificeren van software is een van de centrale onderzoeksgebieden in de informatica. Centrale vraag hierbij is wat voor computerprogramma's de geschikte analogen zijn van plattegronden voor gebouwen of blauwdrukken van machines. Er zijn verschillende specificatietalen voor software ontwikkeld, met verschillende abstractieniveaus. Sommige van deze talen zijn geschikt voor het beschrijven van systemen op hoog abstractieniveau, en handig in de eerdere stadia van de ontwikkeling van een systeem. Anderen talen zijn juist geschikt voor het in detail beschrijven van gedetailleerde ontwerpbeslissingen van de uiteindelijke implementatie, op hetzelfde – relatief lage – abstractieniveau van de gekozen programmeertaal.

Een van de succesvolste lage niveau specificatietalen is de specificatietaal JML voor de populaire programmeertaal Java. Deze taal heeft zich ontwikkeld tot de wereldwijde standaard in de academische gemeenschap. Doordat de specificatietaal gebaseerd is op Java, heeft de taal een solide basis, en is er een groeiende collectie tools die geautomatiseerde ondersteuning bieden bij het controleren van JML specificaties. Een beperking van JML is dat het een laag abstractieniveau heeft – specificaties worden erg lang – en dat het geen methodologie of ondersteuning voor het ontwikkelen en beheersen van groter collecties specificaties.

Doel van dit project is om bij JML een bijbehorende hoge niveau specificatietaal te ontwikkelen, gebaseerd op bestaande hoge niveau specificatietalen met bijbehorend tool support. Dit is een cruciale stap om het gebruik van JML, en daarmee ook het gebruik van de groeiende verzameling tools voor JML, voor grotere softwaresystemen mogelijk te maken,

3) Classification

This proposal involves aspects from many disciplines and sub-disciplines of Computer Science used by NWO:

3. Software Engineering	3.2. Specificatiemethoden
	3.2. Constructiemethoden
	3.4. Testmethoden
	3.6. Ontwikkeltools
6. Fundamenten	6.3. Semantiek
	6.5. Formele methoden

4) Composition of the Research Team

Name	hours per week	specialism
Dr. J.R. Kiniry (postdoc)	6	software engineering; formal methods for reuse; specification languages (esp. JML, BON, & EBON); specification & verification of Java (esp. ESC/Java)
Dr. M. Oostdijk (postdoc)	6	translation of high level specifications to JML; specification & verification of Java
Prof. Dr. B.P.F. Jacobs	2	semantics, specification & verification for Java; use of proof tools (esp. PVS)
Dr.ir E. Poll (UD)	4	program logics and verification tools; formal methods for Java & JML
Project programmer	40	
Project postdoc	40	

5) Research School

The research group is part of the Institute for Programming research and Algorithmics (IPA). The proposed research falls within IPA's main theme 'Formal Methods', more specifically in the sub-themes 'Semantics en Verification' and 'Formal Systems'.

6) Description of Proposed Research

6.1 Background

JML

The Java Modeling Language (JML) is becoming the standard low level specification language for Java. In JML, Java classes can be annotated with class invariants and method pre- and post-conditions. These annotations are added to the Java source code in the form of special comment tags so that the Java compiler can ignore them, comparable to Javadoc tags.

The academic community has a tradition of developing many separate languages by isolated groups. One of the exciting aspects of JML is that it is broadly supported by many international cooperating groups, both within academia and industry (especially for smart cards).

A growing collection of tools is becoming available for JML, for instance the JML runtime checker [5], ESC/Java [9], Daikon [8], and the Loop tool [2], incl. one commercial tool, JACK [4], developed by the smartcard manufacturer Gemplus. For an overview, see [3]. These tools form a spectrum of analysis methods. On one end of the spectrum is runtime checking, which is essentially testing the code. Runtime checking may not give complete or sound answers about the correctness of the program, but it fits very well into the current practice of software engineering. The center of the spectrum consists of tools that do static analysis of the annotated Java code, such as ESC/Java 2. On the far end of the spectrum is theorem

proving with the Loop tool which requires an expert user to perform relatively many interactions with the tool, but gives absolute certainty about the correctness of the program once it has been proved correct.

Note that the group in Nijmegen has plenty of experience with these tools, both in applying them to case studies (e.g. see [26]) and in developing the tools and the underlying theories: the Loop tool was developed in Nijmegen and ESC/Java, whose source code has now been released by Compaq, is being extended by Kiniry et al. Furthermore, the two investigators of this proposal are on the JML language development team.

Extensive experience with JML and its tools shows that getting the low level JML specification right is often the hardest part. JML specifications can easily become as large as the Java program itself, and writing them is a serious challenge, especially since JML offers no methodology for this, and there are no tools that can help in this job. Providing such a methodology and tools is the main challenge, given that several JML tools are now mature enough to be applied in industry, if only more low level JML specifications would be available. To address this challenge, this project will develop a higher level specification language that is better suited to develop and maintain specifications.

Within the now successfully completed VerifiCard project the language JML has been used extensively, also within the participating smart card industries. While these industries are enthusiastic about the benefits of JML for their code development, they see as a major next challenge building the link between their high level informal specifications (e.g., big piles of paper) and their low level JML specification.

High Level Specification Languages

High level specifications typically capture only certain interesting properties of a system. They abstract away from implementation details, and thereby leave more freedom to whomever is supposed to implement the software. High level specifications are usually made before or during development, and are therefore abstract.

Programs are represented by iconographic or textual diagrams which describe different aspects of the system—static structure, dynamic behavior, etc. CASE tools permit the manipulation of the diagrams which, in turn, change the associated source code. Occasionally, the inverse is true—manipulations of the source code cause changes in the diagrams. In most high level representations, the diagram-code refinement relationship is a one-way translation, but in a rare few, it is two-way.

Unfortunately, in the vast majority of high level languages and tools that support them, these relationships and refinements are unspecified, undocumented, and informal. At best, they are hard-coded into the source code of the tools, or are described in ambiguous natural language documentation.

The Unified Modeling Language The Unified Modeling Language (UML) is the *de facto* industry standard for high level specification of object-oriented programs [6, 7]. It is a very large diagram-based notation with around a dozen different kinds of diagrams using hundreds of different artifacts, all for representing different views on the system being specified. UML is a very popular choice for high level specification research because it is used in industry. But because UML is so large and most of the language has no semantics, UML-centric research always uses a small subset of the language.

BON and EBON Rather than take a very large ambiguous language and cut away unnecessary constructs, we instead choose to take a small language which has a well-defined semantics and add necessary constructs. The languages that we have chosen to modify and extend from a JML/Java-centric point-of-view are called BON and EBON. Since our new language is inspired by (E)BON, and focuses on JML and Java, we will call it "JBON".

BON is a high level specification language described in Walden and Nerson's *Seamless Object-Oriented Software Architecture* [28], extended from an earlier paper by Nerson [23]. Because BON is a small high level specification language with a well-defined semantics, it can be thought of as a stable semantic core of UML.

BON is used within several commercial and Open Source tools: Eiffel Software's EiffelStudio IDE; Ehrke's BonBon CASE tool; Steve Thompson and Roy Phillips's BONBAZ/Envision project; Monash University's work on MON, and MONCORE, TROOPER, and YOOCC tools [1, 20, 21]; Kaminskaya's

BON static diagram tool [14]; Paige, Kaminskaya, Ostroff, and Lancaric's BON-CASE tool [25] as well as Paige and Ostroff's extensive recent related work [24, 27]; and Kiniry's EBON tool suite [15].

Over the past few years the BON specification language has been extended with a set of new constructs inherited from work in code standard specifications [16]. This new language is called *Extended BON* (EBON) and the new constructs used to extend BON are called *semantic properties* [18].

Semantic properties are domain-specific high level specification constructs used to augment an existing language with richer semantics. Semantic properties are specified at two levels: loosely with precise natural language, and formally within the semantics of EBON. The refinement relationships between these two specification levels as well as between a semantic property's use in an EBON specification and its realization in program code written in JML-annotated Java is also formally defined.

Basic interpretations from Extended BON to Java and Eiffel program code have been defined in earlier work [17, 18]. We intend to extend this semantics to: (a) integrate the various types of refinement inherent in BON, JML, and Java, and (b) to ensure the coherency of verification under specification and program code changes. Both of these points are discussed below in more detail.

6.2 Proposed Work

We intend to define a high level specification language JBON, appropriate for developing, summarizing, modifying coherently, maintaining and reasoning about JML-annotated Java modules. This work entails specifying JBON's formal semantics by structure-preserving interpretations between JBON and JML-annotated Java, and the semantics of refinement operations inherent in EBON and JML-annotated Java. We will also construct a tool that helps Java and JML developers write, maintain, check, and reason about JML-annotated Java modules at multiple refinement levels.

Serious case studies will be carried out using the language and the associated tools as they are developed, in order to test these and steer development.

JBON JBON will be a higher-level specification language than JML, largely based on the languages of BON and EBON, but tailored to JML in the way that JML is tailored to Java.

Although they can be used to target other OO programming languages, BON and EBON were originally developed for the high level specification of Eiffel programs, and they contain some features which mirror features of Eiffel. To support a smooth translation to Java and JML, JBON will exclude some of the Eiffel-specific notions and terminology that are not present in Java or JML, and will includes features found in Java or JML. To give a concrete example, JBON will use the Java notions of visibility, such as `public` and `protected`, rather than the finer-grained Eiffel notions that BON allows, and JBON should allow the specification of exceptions, which are much more frequently used in Java than in Eiffel [19]. Examples of what JBON specifications might look like are given in Figure 1.

Critical Aspects JBON will be an unusual high level specification language in that it is *seamless* and *reversible*, and will focus on *contracts* and *models*, and its definition will be driven by a low level specification language (JML).

JBON is *seamless* because it is designed to be used during all phases of program development. Multiple refinement levels (high level design with charts, detailed design with types, and dynamism with scenarios and events), coupled with explicit refinement relationships between those levels, means that JBON can be used all the way from domain analysis to unit and system testing and code maintenance.

By virtue of its design, every construct described in JBON is fully realizable in program code. One can specify system structure, types, contracts, events, scenarios, and more. Each of these constructs can not only be translated into program code, but program code can be translated into JBON. Thus, JBON is called a *reversible* specification language. This makes JBON unique insofar as, with proper tool support, a system specification need not become out-of-date if it is written in JBON.

JBON focuses on software *contracts* (pre- and postconditions and invariants) and *models* as the primary means of expressing a system's semantics. Contracts have a well-defined semantics which matches that of other contract-based, object-oriented specification languages such as JML. JBON models provide high level constructs like state charts, collection classes, and algebraic data-types which are mapped directly

down to low level JML models. This makes the language more expressive while retaining a complete semantics, while hiding these complex notions in the lower-level refinements.

Translations between JBON and JML-annotated Java The most important translation in the project will be the translation of JBON to JML-annotated Java. This defines the semantics of JBON, and the implementation for this translation is crucial for allows JBON to be used to develop JML-annotated Java. Development of this translation may suggest the need for refinements of the original JBON language. It may also suggest extensions of JML to capture the relationships, ie. association (has-a) and aggregation (part-of), that can be expressed in JBON. For instance, the JBON notion of aggregation might be captured in JML using a notion of ownership. Informal parts of JBON specification, say a one-sentence description of a class in natural language, cannot be translated to JML but can be translated to corresponding Javadoc tags.

A reverse translation, from Java to JBON will be harder because a single high level JBON construct may well correspond to a collection of several lower-level JML constructs. The main aim of the translation from Java to JBON is to provide a starting point to develop a JBON spec for some existing code or API. Like its inverse, this translation may suggest some additions to JBON (e.g., reverse engineering). For JML, for instance, the need to distinguish which uses of inheritance in Java code are meant to achieve behavioral subtyping and which are just done for code inheritance.

Semantics of JBON As a justification of the translation from BON to JML-annotated Java, and as a guide for the development of suitable notions of refinement for JBON, a semantics for JBON will be developed. The semantics will be a type-theoretic and coalgebraic one, so as to be comparable with our existing work on the semantics of JML and Java [13]. However, we do not envisage a completely formalized semantics as we have developed for JML and Java, as theories for the theorem prover PVS, because it is simply too much work given the requested resources.

Vertical and Horizontal Refinement Notions of refinement are crucial to support the convenient development and maintenance of specification. This includes notions of vertical refinement between specifications at a given point in time, and notions of horizontal refinement, between specifications over time. Inheritance and aggregation are already two notions of vertical refinement. In addition, we also want support for:

- the refinement of informal specifications to formal specifications using the different levels of (in)formality inherent in JBON,
- the factoring out of common part of specification, modulo renaming,
- labeling and naming of parts of specifications,
- changes in visibility and export status, and
- changes in choice of models.

Notions of horizontal refinement support the evolution and changes to specifications and code specifications over time. For example, we want changes to names or visibility of members, either in the JBON or the JML-annotated Java to be automatically propagated, and deletions of, say, an invariant the code to be automatically propagated to the JBON specifications.

Tool support All the features listed above will be implemented in a case tool for JBON. The implementation of this tool will be based on existing open-source tools, namely, for JBON, the EBON case tool (for the textual representation) and the BON-CASE (for the graphical representation), and for JML and Java, the frameworks provided by the JML tool suite and ESC/Java2. We have already developed a small "Auto-JML" tool suite of tools for translation of high level specifications to low level JML specifications [10, 11] which can be reused in the translation from JBON to JML-annotated Java.

Case studies In parallel with the task sketched above, some serious case studies will be investigated with the aim to test the results of these tasks and their implementations. These case studies will include examples for which we already have JML specifications which we want to extend and maintain for future versions, examples for which we only have Java code, and examples for which we have (E)BON specifications but no Java code yet, in order to try out all the purposes for which JBON and its tools can be used.

Potential case studies are

1. JavaCard and OpenPlatform APIs
2. JDK, the core Java API classes for which JML specifications have already been developed by various groups in the JML consortium
3. J2ME API (`http://java.sun.com/j2me/`)
4. EiffelBase [22] or Gobo (`www.gobosoft.com`) collection library
5. Java-based Component Architectures (e.g., JavaBeans)

For 1. and 2. large collections of JML specifications have already been developed, by ourselves and other groups in the JML consortium, and there is a commitment to extend and maintain these for future releases. For 4. on the other hand, BON specifications and Eiffel implementations already exist, but no (Java) implementation has been written as of yet.

An Example Figure 1 is an example JBON specification for a telephone card API, developed as part of a case study performed by our group [12]. The corresponding JML-annotated Java interface is found in Figure 2.

This example shows several interesting aspects of JBON and their refinement to JML and Java. In particular, notice that:

- High level natural language descriptions of methods and invariants in the top-most class chart are directly mapped down into (i) the medium-level JBON specification of the telephone card class (in the Telephone_Card_Design diagram), and (ii) the Javadoc documentation of the resulting Java interface. This means that informal documentation is written and maintained at the appropriate abstraction level, not embedded in and with the program code.
- Indexing expressions denote the structure of the code. E.g., a package property denotes the generated Java package. Other properties are mapped directly to appropriate Javadoc tags.
- High level class annotations like query, deferred, reused, and interfaced (all existing (E)BON keywords) are mapped to (potentially complex) notions at the JML/Java level. E.g., queries are pure methods, deferred classes are abstract, reused classes are fully public, and interfaced classes are Java interfaces.
- Models are declared via model features. This permits the specification of (JML) model fields, methods, and classes and their use in the specifications of non-model features. This is significantly improves the expressiveness of (E)BON.
- (E)BON delta expressions are used to denote frame conditions (modifies clauses) on methods.
- New JBON-specific constructs are available to denote exception types (the caret operator) and specify exceptional postconditions (signals), both necessary for JML/Java.
- Not only can invariants be specified within static diagrams, but JML-inspired initially and constraint clauses are also permitted.

```
class_chart Telephone_Card
indexing
  package: "simplermi";
  author:  "Joe Kiniry";
  created: "January, 2004";
explanation "A small example telephone card-like API."
query
  "Get the balance."
command
  "Set the balance."
  "Decrement the balance by one."
invariant
  "The card always has a non-negative balance."
end

static_diagram Telephone_Card_Design
  -- The Java/JML design of the Telephone_Card class.
component
  deferred class SimpleRMIInterface
  reused interfaced
    -- The generic interface for a Telephone_Card.
  inherit java.rmi.Remote

  model feature
    -- The models of this interface.
    balance : Integer;
    state : { init, issued, locked };

  public feature
    -- The public features of an RMI-enabled Telepone_Card.

    setBalance
      -- Set the balance.
      -> b: short
      ^  UserException, RemoteException
    ensures delta state and delta balance;
    ensures old state = init and b > 0 -> balance = b;
    signals for_all ue : UserException it_holds
           (b <= 0 or old state != init) and balance = old balance;

    getValue: short
      -- Get the balance.
      ^ RemoteException
    ensures Result = balance;

    decValue
      -- Decrement the balance by one.
      ^ UserException, RemoteException
    ensures delta balance and delta state;
    ensures old state = issued and old balance > 1 ->
           state = issued and balance = old balance - 1;
    ensures old state = issued and old balance = 1 ->
           state = locked and balance = 0;
    signals for_all ue UserException it_holds
           old state /= issued and balance = old balance;

  initially
    -- The card starts in the initialization state with zero balance.
    state = init and balance = 0;
  invariant
    -- The card state must be either initialization, issued, or locked.
    state member_of { init, issued, locked };
    -- The card always has a non-negative balance.
    balance >= 0;
    -- If the card is initialization, its balance must be zero.
    state = init -> balance = 0;
    -- If the card is issued, its balance is positive.
    state = issued -> balance > 0;
    -- If the card is locked, its balance must be zero.
    state = locked -> balance = 0;
  constraint
    -- Balance is non-monotonically decreasing.
    old state = issued and state = issued ->
      balance > 0 and old balance >= balance;
end
```

Figure 1: JBON Specifications of a Telephone Card Class

```
package simplermi;

import java.rmi.Remote;
import java.rmi.RemoteException;
import javacard.framework.UserException;

/**
 * A small example telephone card-like API.
 *
 * @author Joe Kiniry
 * @created January, 2004
 **/

public interface SimpleRMIinterface extends Remote
{
  //@ model instance int _balance;
  //@ model instance int _state;
  //@ model final static int _STATE_INIT;
  //@ model final static int _STATE_ISSUED;
  //@ model final static int _STATE_LOCKED;

  /*@ invariant _STATE_INIT != _STATE_ISSUED &&
                _STATE_INIT != _STATE_LOCKED &&
                _STATE_ISSUED != _STATE_LOCKED;
      invariant _state == _STATE_INIT ||
                _state == _STATE_ISSUED ||
                _state == _STATE_LOCKED;
      invariant _balance >= 0
            && ( _state == _STATE_INIT ==> _balance == 0 )
            && ( _state == _STATE_ISSUED ==> _balance > 0 )
            && ( _state == _STATE_LOCKED ==> _balance == 0 );
      constraint _STATE_INIT == \old(_STATE_INIT) &&
                 _STATE_ISSUED == \old(_STATE_ISSUED) &&
                 _STATE_LOCKED == \old(_STATE_LOCKED);
      constraint
         ( _state == _STATE_ISSUED && \old(_state) == _STATE_ISSUED ) ==>
         0 < _balance  && _balance <= \old(_balance); */

  /*@ behavior
        requires true;
        ensures (\old(_state) == _STATE_INIT && b > 0) ==> _balance == b;
        signals (UserException ue)
           ( b <= 0 || \old(_state) != _STATE_INIT )
           && _balance == \old(_balance);
        signals (RemoteException re) true; */
  public void setBalance(short b) throws UserException, RemoteException;

  /*@ behavior
        requires true;
        assignable \nothing;
        ensures \result == _balance;
        signals (RemoteException re) true; */
  public /*@ pure @*/ short getValue() throws RemoteException;

  /*@ behavior
        requires true;
        assignable _balance;
        assignable _balance;
        ensures (\old(_state) == _STATE_ISSUED && \old(_balance) > 1) ==>
                (_state == _STATE_ISSUED && _balance == \old(_balance) - 1)
             && (\old(_state) == _STATE_ISSUED && \old(_balance) == 1) ==>
                (_state == _STATE_LOCKED && _balance == 0);
        signals (UserException ue)
           ( \old(_state) != _STATE_ISSUED )
           && _balance == \old(_balance);
        signals (RemoteException re) true; */
   public void decValue() throws UserException, RemoteException;

   ...
}
```

Figure 2: Excerpt of JML-annotated Java Telephone Card Interface

Related Work Initial work exists extending the BON-CASE tool [25] to JML, using the translation from BON to JML described in [14]. However, this work uses "pure" BON as the high level language, and restricts the support for Java and JML accordingly, whereas we will use JML and Java as the starting point, and adapt BON accordingly. This choice means we can take advantage of the broad existing tool-base for JML and we will not be restricted by the existing JML-independent semantics of BON.

Embedding It should be noted that the existing *Pionier* grant ("Program Security and Correctness") serves as a background for this work. However, the current proposal has a clear software engineering focus, as opposed to the security focus of the *Pionier* project, and goes well beyond that which is planned (and being done) within *Pionier*. Thus, this work will complement, but not duplicate, *Pionier* research.

7) Work Program

Below is a coarse scheduling for the main tasks in the project:

1 Definition of JBON. Month 0-6.

This initial JBON definition will probably be subject to revisions during the remainder of the project.

1a Implementing tool support for JBON. Month 0-12.

2 Translations between JBON and JML-annotated Java. Month 6-12.

2a Implementing tool support for JBON-Java translations. Month 12-24.

3 Semantics of JBON. Month 24-36

4 Vertical and horizontal refinement. Month 12-24

4a Implementing tool support for refinement. Month 18-30

5 Case studies. Month 6-36

Resources

- One Scientific Programmer.
- One Postdoc.
- Portions of time from Dr. Jacobs, Dr. Kiniry, Dr. Oostdijk, and Dr. Poll.

8) Expected Use of Instrumentation

None.

9) Literature

The five key publications for this work are [3, 10, 11, 12, 17].

[1] Jon Avotins, Christine Mingins, and Heinz Schmidt. Yes! an object-oriented compiler compiler (YOOCC). Technical report, Department of Software Development, Monash University, 1996.

[2] J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Software (TACAS)*, volume 2031 of *LNCS*, pages 299–312. Springer-Verlag, 2001.

[3] 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. In Thomas Arts and Wan Fokkink, editors, *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, volume 80 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 73–89. Elsevier, June 2003.

[4] Lilian Burdy, Antoine Requet, and Jean-Louis Lanet. Java applet correctness: A developer-oriented approach. In D. Mandrioli K. Araki, S. Gnesi, editor, *FME 2003*, volume 2805, pages 422–439. Springer-Verlag, 2003.

[5] Y. Cheon and G.T. Leavens. A Runtime Assertion Checker for the Java Modeling Language (JML). In H.R. Arabnia and Y. Mun, editors, *International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada*, pages 322–328. CSREA Press, 2002.

[6] Rational Software Corporation et al. *UML Notation Guide, version 1.1*. The UML 1.1 Consortium, September 1997.

[7] Rational Software Corporation et al. *UML Semantics, version 1.1*. The UML 1.1 Consortium, September 1997.

[8] M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEETSE*, 27(2):99–123, 2001.

[9] C. Flanagan et al. Extended static checking for Java, 2002.

[10] E. Hubbers and M. Oostdijk. Generating JML specifications from UML state diagrams. In *Proceedings of the Forum on specification & Design Languages (FDL 2003)*, pages 263–273. ECSI, 2003.

[11] E. Hubbers, M. Oostdijk, and E. Poll. From finite state machines to provably correct Java Card applets. In D. Gritzalis et al., editors, *Proceedings of the 18th IFIP Information Security Conference*, pages 465–470. Kluwer Academic Publishers, 2003.

[12] B. Jacobs, M. Oostdijk, and M. Warnier. Source code verification of a secure payment applet. *Journal of Logic and Algebraic Programming*, 2003. Special Issue on Smart Cards, To appear.

[13] Bart Jacobs and Erik Poll. Coalgebras and Monads in the Semantics of Java. *Theoretical Computer Science*, 291(3):329–349, 2003.

[14] Liliya Kaminskaya. Combining tools for object-oriented software development: An integration of BON and JML. Master's thesis, Department of Computer Science, York University, 2001.

[15] Joseph R. Kiniry. The Extended BON tool suite, 2001. Available via `http://ebon.sourceforge.net/`.

[16] Joseph R. Kiniry. The KindSoftware coding standard. Technical report, KindSoftware, LLC, 2001. Available via http://www.kindsoftware.com/.

[17] Joseph R. Kiniry. *Kind Theory*. PhD thesis, Department of Computer Science, California Institute of Technology, 2002.

[18] Joseph R. Kiniry. Semantic properties for lightweight specification in knowledgeable development environments. Submitted for publication, 2002.

[19] Joseph R. Kiniry. Exceptions in Java and Eiffel: Two extremes in exception design and application. In *Proceedings of Workshop on Exception Handling in Object Oriented Systems (ECOOP 2003)*, Darmstadt, Germany, July 2003.

[20] Glen Maughan and Jon Avotins. A meta-model for object-oriented reengineering and metrics collection. Technical Report TR95-34, Department of Software Development, Monash University, 1995.

[21] Glenn Maughan and Bohdan Durnota. MON: An object relationship model incorporating roles, classification, publicity, and assertions. Technical report, Department of Software Development, Monash University, 1994.

[22] Bertrand Meyer. *Reusable Software: The Base Object-Oriented Component Libraries*. The Object-Oriented Series. Prentice-Hall, Inc., 1994.

[23] Jean-Marc Nerson. Applying object-oriented analysis and design. *Communications of the ACM*, 35(9):63–74, September 1992.

[24] R.F. Paige and J.S. Ostroff. ERC: an object-oriented refinement calculus for Eiffel. To appear in Formal Aspects of Computing, 2004.

[25] Richard Paige, Liliya Kaminskaya, Jonathan Ostroff, and Jason Lancaric. BON-CASE: An extensible CASE tool for formal specification and reasoning. *Journal of Object Technology*, 1(3), 2002. Special issue: TOOLS USA 2002 Proceedings. Available online at http://www.jot.fm/.

[26] E. Poll, J. van den Berg, and B. Jacobs. Formal specification of the Java Card API in JML: the APDU class. *Computer Networks*, 36(4):407–421, 2001.

[27] Ali Taleghani and Jonathan S. Ostroff. The BON development tool. In *Proceedings of the Eclipse Technology eXchange eTX/OOPSLA'03*, Anaheim, CA, October 2003.

[28] Kim Waldén and Jean-Marc Nerson. *Seamless Object-Oriented Software Architecture - Analysis and Design of Reliable Systems*. Prentice-Hall, Inc., 1995.

10) Requested Budget

		Salary	Bench fee
Scientific Programmer	3 yr	173.910 EUR	0.0 EUR
Postdoc	3 yr	162.227 EUR	4.538 EUR