ISSN 2186-7437

NII Shonan Meeting Report

No. 2013-3

The Java Modeling Language (JML)

Gary T. Leavens
University of Central Florida, Orlando, FL, USA
Peter H. Schmitt
Karlsruhe Institute of Technology, Karlsruhe, Germany
Jooyong Yi
National University of Singapore, Singapore

May 13–16, 2013

National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-Ku, Tokyo, Japan

The Java Modeling Language (JML)

Organizers:
Gary T. Leavens, (University of Central Florida, Orlando, FL, USA)
Peter H. Schmitt (Karlsruhe Institute of Technology, Karlsruhe, Germany)
Jooyong Yi (National University of Singapore, Singapore)

May 13–16, 2013

Topic of the Meeting

Program verification has been a topic of research interest far into the history of computing science. Today, it is still a key research focus, see e.g., Hoare's Verified Compiler Grand Challenge and the Verified Software Initiative, whose flagship activities are the series of VSTTE workshops (Verified Software: Theory, Tools, and Experiments) and the launch of a series of verification competitions. A main facet in this effort is the ability to formally express properties that must be verified. Building on a long line of work in formal methods for reasoning about behavioral specifications of programs, several recent languages balance the desire for completeness and the pragmatics of checkability. In the context of the object-oriented programming paradigm, the Java Modeling Language (JML) is the most widely-adopted specification language in the Java formal methods research community.

The Java Modeling Language (JML) is a formal, behavioral specification language for Java. It describes detailed designs of Java classes and interfaces using pre- and postconditions, invariants, and several more advanced features. JML is used as a common language for many research projects and tools, including a runtime assertion checker (jmlc), tools to help unit testing (jmlunit), an extended static checker (ESC/Java), and several formal verification tools (e.g., LOOP, JACK, KRAKATOA, Jive, and KeY). JML is seeing some use in industry, particularly for financial applications on Java smart cards and for verifying some security properties of a computer-based voting system.

Since JML is widely understood in the formal methods research community, it provides a shared notation for communicating and comparing many advances, both theoretical and practical, and it serves as a launching pad for research on advanced specification language features and tools. Researchers are using JML to study or express results for a wide variety of problems; these problems include verification logics, side effects (including frame axioms and modifies clauses), invariants, behavioral subtyping, null pointer dereferences, interfacing with theorem provers, information hiding, specifying call sequences in frameworks, multithreading, compilation, resource usage, and security. In addition to the tools mentioned above, JML is also used to express, compare, or study tools for checking specifications, unit testing, and specification inference. JML is used to state research problems for formal specification languages and for general discussions of specification language design. JML has also inspired at least three other similar specification languages, Spec#, BML, and Pipa, and has influenced the design and tools for Eiffel. Representatives of these communities are included in the invitation list. JML tools are used in the implementation of at least two other specification languages: ConGu and Circus. At present, there are at least 19 research groups around the world that are cooperating on

JML-related research. These groups, and others, have published over 200 papers directly related to JML (see `http://www.jmlspecs.org/papers.shtml`).

Participants

1. Bernhard Beckert, Karlsruhe Institute of Technology, Karlsruhe, Germany
2. Stefan Blom, University of Twente, Enschede, The Netherlands
3. Daniel Bruns, Karlsruhe Institute of Technology, Karlsruhe, Germany
4. Richard Bubel, Technische Universität Darmstadt, Darmstadt, Germany
5. Néstor Cataño, The University of Madeira, Funchal, Portugal
6. Patrice Chalin, Kansas State University, Manhattan, KS, USA
7. David Cok, GrammaTech, Ithaca, NY, USA
8. Werner Dietl, University of Washington, Seattle, WA, USA
9. Reiner Hähnle, Technische Universität Darmstadt, Darmstadt, Germany
10. Marieke Huisman, University of Twente, Enschede, The Netherlands
11. Gary T. Leavens, University of Central Florida, Orlando, FL, USA
12. K. Rustan M. Leino, Microsoft Research, Redmond, WA, USA
13. Wojciech Mostowski, University of Twente, Enschede, The Netherlands
14. David Naumann, Stevens Institute of Technology, Hoboken, NJ, USA
15. Robby, Kansas State University, Manhattan, KS, USA
16. Peter H. Schmitt, Karlsruhe Institute of Technology, Karlsruhe, Germany
17. Robert Wille, University of Bremen, Bremen, Germany
18. Jooyong Yi, National University of Singapore, Singapore
19. Daniel Zimmerman, University of Washington Tacoma, Tacoma, WA, USA

Overview of Talks

JMLUnitNG: Present and Future

Daniel Zimmerman, University of Washington Tacoma, USA

JMLUnitNG is an automated test framework for JML-annotated Java code, developed as an improvement over the original JMLUnit test framework, and has been publicly available since 2010. It has several advantages over the original JMLUnit, including much improved memory efficiency, the ability to run far more tests without requiring users to edit source code, and the ability to work with Java code written using language features introduced in Java 1.5 and later.

This talk describes and demonstrates the current status and capabilities of the JMLUnitNG tool, and introduces new features and improvements planned for future versions of the tool; the latter include a graphical interface and IDE plug-in, better test generation techniques, better feedback for users, and a public source code release.

The EventB2Java Tool for Generating JML-Specified Java Implementations of Event-B Models

Néstor Cataño, The University of Madeira, Portugal

I presented the EventB2Java tool that produces JML-specified Java implementations of Event-B models. We implemented EventB2Java as a plug-in of the Rodin platform. I have validated EventB2Java by generating Java code for several Event-B models, including a moderately complex model for social networking, and an EventB model of the Binary Search algorithm.

Verification of JML Expressions Using UML/OCL-based Approaches

Robert Wille, University of Bremen, Germany

At first glance, the Java Modeling Language (JML) and the Object Constraint Language (OCL) share many similarities. However, thus far, solutions for testing and verification of the corresponding specifications have been developed independently of each other. In the talk, we discussed possible ways to exploit the knowledge and the solutions of the respective research areas. We proposed an initial scheme for the mapping of JML to UML/OCL specifications. The proposed mapping is conducted in two steps:

First the Java code is translated into a UML class diagram representing the structure of the program.

Afterwards the JML constraints are mapped to the corresponding OCL expressions forming restrictions and constraints on the program.

Initial case studies showed how, using this scheme, JML specifications can be verified using UML/OCL verification approaches.

Demo of the KeYSystem

Peter H. Schmitt, Karlsruhe Institute of Technology, Germany

Using a small example, comprising three methods, this demonstration provided a detailed walk through the various artefacts used in the program verification process of the KeY system:

- JML annotated Java source file,
- proof obligation browser,
- proof obligations formalized in Dynamic Logic
  (this is the internal logic of the KeYsystem)
- first-order proof obligations

JML was – obviously – assumed to be known to the audience. Short one-slide introductions to Dynamic Logic, sequents, and sequent calculus rules were provided along, and in addition the formal representation of proof rules was shown, called *taclets* in the KeY vernacular.

The example verification task included the use of the abstract data type of finite sequences and the notation of permutation of finite sequences. A glimpse into the taclet axiomatization of this data types was given.

OpenJML: Status and Challenges

David Cok, GrammaTech, Ithaca, NY, USA

OpenJML is a set of tools supporting the Java Modeling Language (JML) for Java 1.7. Though the implementation is not complete, OpenJML supports the basic level 0 features of JML and many higher level features. The overall goal of OpenJML is to provide an encoding of Java and JML into logical assertions in SMTLIBv2 format that can be submitted to off-the-shelf SMT solvers for validation or refutation. By this means, the JML community can explore a number of research avenues: the utility of novel specification language features; the effect of styles of writing annotations on verification success; the effect of language encoding (into SMT) on verification success; the abilities (and room for growth) of SMT solvers as applied to software verification. The OpenJML tool set includes both a command-line tool and an Eclipse plug-in; the GUI implementation and the overall user-facing feature set is intended to make verification-style formal methods accessible in educational settings, and the author invites collaboration to ensure the tool has the features needed for use in the classroom in the fall of 2013. In particular, the tool provides warning reports and counterexample information in the context of Java source code, much as a developer might use any IDE. OpenJML is intended to replace ESC/Java2, which only supports Java 1.4, by leveraging an existing and active Java compiler: the tool is built on the OpenJDK compiler toolset for Java, so that it can readily incorporate bug fixes and new features as Java develops.

Ghost Code and Algorithmic Specification in the Java Modeling Language

Bernhard Beckert, Karlsruhe Institute of Technology, Germany

The standard approach to specification with JML is to declaratively describe the states that Java programs reach using pre-/post-condition pairs and invariants. But in many cases, it is also important to be able to describe the behaviour of Java programs algorithmically by defining an abstract program or automaton and stating what the relation between the abstract program and the concrete Java program is. Currently, the way to do this with JML is to use model fields and ghost code. In my talk, I propose additional features for JML that give more support for abstract algorithmic specifications.

One useful extension are abstract data types. Another one is an operator `choose` that allows to write non-deterministic ghost code, where the source of non-determinism is made explicit in the code instead of being hidden in the definition of model fields. It should then also be allowed in JML to define *ghost methods* that are non-deterministic and are allowed to (only) change ghost state.

To describe the relation between (abstract) ghost methods and concrete Java code, I propose to introduce the modal operators `[]` (box) and `<>` (diamond) known from dynamic logic to JML. This would, for example, allow to write

```
invariant
  \forall int res; ( <concreteM()> \result == res ==>
                     <abstractM()> \result == res
                   )
```

to express that `concreteM()` refines `abstractM()`.

To illustrate the usefulness of these extensions, I present the specification of a Java program implementing the Single Transferable Vote election scheme.

Generics: an Annotation Preserving Type Erasure

Stefan Blom, University of Twente, The Netherlands

In this talk, we present ongoing work which aims to verify Java programs that use generics and have been specified by transforming them into a specified Java program without generics and then verifying that simplified program.

This transformation will be used in the VerCors toolset to verify concurrent Java programs and libraries that have been specified using JML extended with separation logic. Among them `java.util.concurrent`, the Java concurrency package. The VerCors toolset works by translating specified Java program into Chalice programs. The latter does not support generics, so we need a transformation that erases generics. Another use of the transformation is as a pre-processing step for using JML tools do not support generics.

The effect of erasure on the code of the program is essentially the same transformation that a Java compiler uses. The transformation for specifications is more difficult: generics annotations contain information about which objects are instances of which classes. This information must be transformed into additional (generics free) specifications because otherwise static checkers will no longer be able to successfully verify the transformed program.

Reuse in JML by Specification Deltas and Abstract Contracts

Reiner Hähnle, Technische Universität Darmstadt

Modern software tends to undergo frequent requirement changes and typically is deployed in many different scenarios. This poses significant challenges to formal software verification, because it is not feasible to verify a software product from scratch after each change. It is essential to perform verification in a modular fashion instead. The goal must be to reuse not merely software artifacts, but also specification and verification effort.

In our setting code reuse is realized by delta-oriented programming, an approach where a core program is gradually transformed by code "deltas" each of which corresponds to a product feature. The delta-oriented paradigm is then extended to JML specifications and to verification proofs. As a next step towards modular verification we transpose Liskov-Leavens' behavioural subtyping principle to the delta world. Finally, based on the resulting theory, we perform a syntactic analysis of contract deltas that permits to automatically factor out those parts of a verification proof that stays valid after applying a code delta. This is achieved by a novel verification paradigm called "abstract verification". A few simple language concepts are sufficient to make JML support specification and verification reuse.

For Programs and Proofs: Mo' Specs and Mo' Math

K. Rustan M. Leino, Microsoft Research, Redmond, USA

Behavioral program specifications can be given at different levels of details. As we increasing our ambitions to write more detailed specifications, we need more features in the specification language. These features include a convenient and user-extensible repertoire of types (like mathematical integers, sets, sequences, and inductive datatypes), ghost constructs (which look like ordinary program constructs but are specification-only and are ignored by the compiler, like ghost variables, ghost parameters, and ghost methods), and user-defined, and often recursive, functions. When specifications become more involved, so do proofs. Thus, to support proofs of more expressive specifications, proof systems will need a combination of more automation and more user-defined proof guidance. For example, such guidance can include user-supplied declarative proofs or proof sketches.

In this presentation, I give a taste of how these considerations have played out for the verification-aware programming language Dafny.

From JML to Spark and Back

Patrice Chalin, Kansas State University, USA

At the last JML meeting in Dagstuhl (2009), some of the key topics discussed included: an adoption of strong validity for assertion semantics, the JML Intermediate Representation (JIR), Java Contracts and possible use of JML 5

annotations, i.e., the Meta Data Facility (MDF), as the principle means of encoding JML specification constructs. Discussions of these topics was lead by myself and Robby from K-State's SAnToS Lab.

In terms of what is relevant to the JML community, SAnToS lab is currently leading two main research thrusts: the Medical Device Coordination Framework (MDCF) and the Sireum Kiasan symbolic execution tools. In the context of the MDCF, use of Model Driven Engineering (MDE) techniques are being explored through use of the Architectural Analysis and Design Language (AADL). A SAnToS PhD student is leading the development and maturation of the Behavioral Language for Embedded Systems with Software (BLESS). BLESS can be thought of as a BISL for AADL. In the context of MDE we will also be exploring code generation from BLESS annotated AADL models into SPARK annotated ADA code. Version 1 of Kiasan, the SAnToS symbolic execution tool originally supported JML and then SPARK. Since 2010, Robby has been leading a complete rewrite of the Sireum core framework on which Kiasan is based. Thus there has been a shift in BISL from JML to SPARK, the main reasons for this are that SPARK is considerably simpler than JML and yet is actively used by industry, particularly in (embedded) safety and security critical applications.

SAnToS is also involved in the language design team of the nextgen of SPARK called SPARK 2014. This new edition of SPARK results from: the advent of Ada 2012, lag in the development of Classic SPARK tooling, and the success of the HI-LITE project lead by AdaCore, Altran-Praxis and INRIA. Ada 2012, the latest standard of the Ada language introduces: aspects (akin to Java 5 annotations, but allowing arbitrary syntax to be used in annotation expressions), native support for basic subprogram contracts (via the Pre and Post aspects), type invariants, etc. HI-LITE project goals were to create a new tool architecture where a compiler and verifier would be based on the same front-end, and to explore the possible synergy of combining test and proof. Why? Because unit testing is one of the most expensive parts of the certification of airborne computerized systems. The latest edition of DO-178, namely DO-178C, permits the use of formal methods to discharge some obligations instead of testing. HI-LITE defined the Alfa language, a "safe" subset of Ada 2012 with extra features useful for describing contracts and tests. Like JML, Alfa assertion semantics is based on strong validity and supports arbitrary precision numeric types. SPARK 2014 has been: inspired by Alfa but will retain many of the classic SPARK features and yet will go far beyond the limitations of Classic SPARK. SPARK 2014 development is a joint effort between AdaCore and Altran-Praxis. In a consulting role, I have participated in the SPARK 2014 language design meetings, hoping to share some of JML's growing pains so that problems can be averted.

While some of the features of JML have influenced, and been adopted by, Alfa (and hence the SPARK nextgen), the most important observation we can make about SPARK nextgen, is that the availability of a key native feature like Ada aspects will likely be very beneficial to SPARK. We believe that this would be true of JML as well. Maybe it is time for a JSR proposing an extension to the Java Meta Data Facility.

Explicating Symbolic Execution (xSymExe): An Evidence-Based Verification Framework

Robby, Kansas State University, USA

Previous applications of symbolic execution (SymExe) have focused on bug-finding and test-case generation. However, SymExe has the potential to significantly improve usability and automation when applied to verification of software contracts in safety-critical systems.

Due to the lack of support for processing software contracts and ad hoc approaches for introducing a variety of over/under-approximations and optimizations, most SymExe implementations (and many static analysis approaches) cannot precisely characterize the verification status of contracts (e.g., indicating when contracts are truly verified or violated). Moreover, these tools do not provide explicit justifications for their conclusions, and thus they are not aligned with trends toward evidence-based verification and certification.

To provide a foundation for using SymExe for verification in the context of high-assurance software development, we introduce the concept of Explicating Symbolic Execution (xSymExe) that:

1. builds on a strong semantic foundation,
2. supports full verification of rich software contracts,
3. explicitly tracks where over/under-approximations are introduced or avoided,
4. precisely characterizes the verification status of each contractual claim, and
5. associates each claim with explications for its reported verification status – a detailed presentation of evidence that justifies each claim's reported verification status.

We provide an open source implementation of xSymExe in Bakar Kiasan – a verification tool for the Spark subset of Ada designed for developing critical applications, and we report on case studies in the use of Bakar Kiasan.

A Case Study in Formal Verification Using Multiple Explicit Heaps

Wojciech Mostowski, University of Twente, The Netherlands

In the context of the KeY program verifier and the associated Dynamic Logic for Java we discuss the first instance of applying a generalised approach to the treatment of memory heaps in verification. Namely, we allow verified programs to simultaneously modify several different, but possibly location sharing, heaps. In this paper we detail this approach using the Java Card atomic transactions mechanism, the modelling of which requires two heaps to be considered simultaneously – the basic and the transaction backup heap. Other scenarios where multiple heaps emerge are verification of real-time Java programs, verification of distributed systems, modelling of multi-core systems, or modelling of permissions in concurrent reasoning that we currently investigate for KeY. On the

implementation side, we modified the KeY verifier to provide a general framework for dealing with multiple heaps, and we used that framework to implement the formalisation of Java Card atomic transactions. Commonly, a formal specification language, such as JML, hides the notion of the heap from the user. In our approach the heap becomes a first class parameter (yet transparent in the default verification scenarios) also on the level of specifications.

JML & Specifications Involving Abstract Domains (Preliminary Ideas and Experiences)

Richard Bubel, TU Darmstadt, Germany

In the talk we presented our work in using JML as an intermediate exchange format between a static resource analyzer (COSTA) and a verification tool (KeY) to achieve formally verified resource guarantees of Java programs. In more detail: Program properties that are automatically inferred by static analysis tools are generally not considered to be completely trustworthy, unless the tool implementation or the results are formally verified. Here we focus on the formal verification of *resource guarantees* inferred by automatic cost analysis. Resource guarantees ensure that programs run within the indicated amount of resources which may refer to memory consumption, to number of instructions executed, etc. In previous work we studied formal verification of inferred resource guarantees that depend only on integer data. In realistic programs, however, resource consumption is often bounded by the size of *heap-allocated* data structures. Bounding their size requires to perform a number of structural heap analyses. The contributions of our work are

1. to identify what exactly needs to be verified to guarantee sound analysis of heap manipulating programs,
2. to use and extend JML such that the intermediate analyses results of COSTA can be expressed as assertions and contracts in form of JML annotations,
3. to provide a suitable extension of the program logic used for verification to handle structural heap properties in the context of resource guarantees,
4. to improve the underlying theorem prover so that proof obligations can be automatically discharged.

Exploring info Flow Extensions in a Project that Targets Android Apps

David Naumann, Stevens Institute of Technology, USA

The first part of my talk was an overview of a project seeking tools and techniques for cost-effective evaluation of the trustworthiness of mobile apps. We focus on 'enterprise scenarios', involving mission-related apps and/or access enterprise networks, where incentives and resources exists for substantive evaluations. Our goals are to

1. find flexible and expressive ways to specify information flow requirements for apps,
2. find effective ways to specify what is assumed about the Android platform API, and
3. find practical static analysis and verification techniques to check security of apps with respect to given policies and the platform API.

In our approach, policies have two parts.

(0) A baseline security policy is specified by conventional clearance-level labels on channels; these are propagated through the app's code by type inference, which we plan to implement using the Checker Framework discussed at this meeting.

(1) Downgrading policies are specified in terms of assertions and special relational formulas, to express 'what' part of sensitive data is downgraded (e.g., aggregates, or encryptions) and 'under what conditions' downgrading is allowed (e.g., after user authentication, or after key expiry). Checking of downgrading policies requires conventional assertion checking, for which we are using OpenJML, and relational checking which is being prototyped using Why3 and will eventually be incorporated into a version of OpenJML with extensions to express relational properties. Relational properties subsume information flow properties of a single program (noninterference), as well as equivalence or refinement between programs expressed in terms of coupling relations.

In the second part of this talk I discussed unpublished work on reasoning about relational properties, motivated by equivalence of data representations. I gave a quick sketch of Relational Region Logic, in which dynamic frames are used to express encapsulation boundaries, and a Reynolds-style 'abstraction theorem' appears as a proof rule for program linking. The rule serves to lift a simulation between two ADT implementations to a simulation for a client context linked to the two implementations. I also described how the two programs to be related may be interwoven to facilitate modular verification using intermediate relational assertions.

Proving Information Flow Security with JML and KeY

Daniel Bruns, Karlsruhe Institute of Technology, Germany

There have been static security enforcement techniques based on syntax or types for a long time. While static checking of security type systems provides an attractive and efficient means to enforce non-interference, it is often overly conservative in practice. The reason is that type-based techniques cannot take functional properties into account. For example, a program like `low = high * 0;` is secure, but to verify this one needs to reason about the functionality of `*`. Similarly, to verify that `if (high) {low = f1(low);} else {low = f2(low);}` is secure, one has to verify that `f1()` and `f2()` compute the same.

In contrast, functional program verification techniques tend to be very precise; they can handle the above examples. Many security properties can be defined in this way; the most widely used is *non-interference*: If any two runs

start with the same public inputs, they must agree on the public outputs. In other words, the secret inputs must not influence public outputs. It is an appealing feature of this methodology that the formalization is in most cases a straightforward transcription of the informal definition. Another advantage is the possibility to use existing program verification systems and theorem provers to support verification of the specified properties.

We have extended JML method contracts with a `separates` clause. It specifies a set of locations V, such that the locations *not* in V do not interfere with locations belonging to V. This means that an attacker which can observe one of those sets of locations won't be able to deduce more information through the execution of the method than he or she already knew. This specification can be further refined using declassification. A `\declassify` statement declares an expression which value an attacker may additionally learn. The extension of JML integrates seamlessly with functional JML specifications. This is important since a real precise calculation of information flow dependencies can only be achieved with knowledge on the functional behavior of a program or method. This also works the other way around: knowledge on information flow dependencies does improve functional verification.

Logical information flow analysis has for the greatest part been investigated for simple imperative programming languages. In an object-oriented context the usual definition of low-equivalence of states requiring that the observed values be equal in both states is too strong. Instead only primitive observed values are required to be equal, the observed object values need only be related through a partial isomorphism.

Session on JML in Academic Education

chaired by Marieke Huisman, University of Twente, The Netherlands

We discussed the possibility to integrate the use of JML in the bachelor program. If we want to achieve something in this respect, we need: good tool support, and good text material that introduces the use of annotations in a natural way. We discussed where in the bachelor program this could be integrated. Some participants felt the concern that Java in itself is already too complicated for many first year students (and thus, adding JML would add even more complexity). Probably an algorithms and data structure course would be the most appropriate place to integrate this into the program. Good tool support is currently under development (and will focus on educational use), and as a next step, existing text book material should be collected, and combined and extended into one good tutorial text.

Type Annotations, the Checker Framework, and JML

Werner Dietl, University of Washington, USA

I present my research project to combine pluggable type checking with behavioral interface specifications, in the context of the Checker Framework and JML.

Java currently only supports annotations on declarations, for example, on a class declaration. Type annotations extend Java to allow annotations on all

uses of types in source code. This allows the use of type annotations in Java to implement pluggable type checking. I implemented type annotations in the OpenJDK Java compiler and they will be included in the Java 8 release.

The Checker Framework uses type annotations to provide the infrastructure for pluggable type checkers. A few common type systems are included; for example, the Nullness Checker allows the programmer to ensure null-pointer exception freedom. In recent work, we extended the simple type system infrastructure with a general dataflow framework that supports the specification of pre- and post-conditions of methods. Specialized annotations support particular type systems to specify finer-grained conditions; e.g. in the Nullness Checker, an annotation specifies that a certain field has to be non-null when a method is called. Still, static type systems are conservative in nature and cannot express all possible scenarios — leading to undesirable false positives.

Instead of tailor-making pre- and post-condition annotations for each pluggable type system, I propose integrating pluggable type checking with behavioral interface specifications. The Checker Framework is used to enforce properties that are expressible as type systems. JML specifications are used to refine these type systems with fine-grained specifications. The runtime assertion checker and static verification tools can rely on the guarantees provided by the type systems and only need to check or verify the finer-grained properties not guaranteed by the type systems. Meta-annotations on the type qualifiers are used to specify the translation between type system constructs and the corresponding JML constructs.

By studying the formal relationship between pluggable type checking and more expressive verification approaches, and by providing practical tools, I want to enable developers to go from light-weight type systems to full verification within a common framework. For more details about type annotations and the Checker Framework see `http://types.cs.washington.edu/`.

On the Specification of the Past

Jooyong Yi, National University of Singapore, Singapore

Despite the promise of DBC (Design by Contract) and the advancements of specifications and verifications, DBC has not been adopted to mainstream software development practice yet. I believe that this is largely due the fact that the current specification methods are not cost effective. Most programmers are reluctant to write sophisticated specifications for each method (or function) even at the promise of full verification. Meanwhile, not many programmers are keen on writing simple contracts because intentions behind them can be relatively easily checked (at least partially) by other means like traditional testing. However, all of these do not mean that specifications are useless. To check something, one first need to express the intentions. In this talk, I present two directions of efforts that can increase the cost-effectiveness of specifications.

To make specifications cost-effective, one obvious way is to improve a specification language so that complicated properties can be expressed in an intuitive and succinct manner. For example, I introduced with my colleagues a *past* expression as an alternative to the conventional *old* expression to do structural comparison in a much more intuitive and succinct way.

Another way to improve the cost-effectiveness of specifications is to find a new application domain other than the traditional program verification. For example, I introduced with my colleagues at NUS a *change contract* that describes behavioral and structural changes across program versions. A change contract is typically small because only changes are specified assuming that the rest of behaviors are preserved. Nevertheless, having a change contract can make a big impact on software maintenance because with the intentions spelled out in a change contract, programmers can effectively deal with various software maintenance problems such as wrong bug fixes, regression errors and test case breakage.

Both of the above work were realized as extensions of JML. Also, runtime-assertion checking support was implemented and tested in the OpenJML framework.

Incorporating Region Logic and Separation Logic into JML Framing

Gary T. Leavens, University of Central Florida, Orlando, FL, USA

Framing is important for languages such as JML to allow properties to be carried across method calls when doing verification. JML already incorporates an approach to framing, based on Leino's data groups and the Universe type system. However, in recent years, several new and powerful approaches to framing have arisen, including separation logic, region logic, and the dynamic frames technique. In this talk we propose an extension to JML that incorporates primitives necessary to support region logic and the dynamic frames technique directly. We also describe a translation from separation logic into this extension and argue that general separation logic specifications can be translated without change of meaning into this extended JML. Thus this extension would allow different styles of specifying frames to be written in JML, but supported by a common semantics in tools.

This represents joint work with Yuyan Bao and was supported in part by the US NSF under grant CCF-0916715.

Conclusion

After introductions, Gary Leavens lead a discussion about the current state of JML and what he called "JML's Bright Future". In staking this position, he said that he believes that Java will be important for a long time and that by working together all the participants in the JML project can lower costs for users and maximize user benefits.

The advantage of JML is that it acts as *glue* for formal methods tools and lets users apply many different tools to a single specified piece of code. He characterized the mission of the JML effort as:

- To make formal methods for Java practical and useful, and
- To enhance our individual reputations and careers.

Gary enumerated several successes of the JML project since the first days in the late 1990s, but emphasized that the successes were largely due to having useful tools. He said that the "way forward" for JML was through the OpenJML platform, and urged all concerned to work towards the success of OpenJML.

In the final discussion session on the Future of JML, Gary Leavens started by noting that JML is at life stage that Wynn's paper ("Organizational Structure of Open Source Projects: A Life Cycle Approach," 7th Annual Conference of the Southern Association for Information Systems, Georgia, 2003) calls "Decline or Revival". He committed himself to JML's revival and urged the participants to also help revive JML. He committed himself to working on the JML Language definition and a JML textbook (a need pointed out in the education discussion). He called for volunteers from the JML community to work on the following topics:

- Maintain parts of web site Test OpenJML and provide bug reports
- Submit patches for bugs
- Provide (shell) scripts, packaging for OpenJML

He also urged the participants to get their students involved in:

- Testing and filing bug reports on OpenJML
- Submitting patches for bugs

And he urged the participants to volunteer to recruit such students, to coordinate documentation about OpenJML, to cooperate with David Cok to carve out pieces of OpenJML for the most promising volunteers (with a view towards making them active developers), and to publicly reward and encourage volunteers.

On the whole the organizers and participants were very pleased and happy with the workshop. Participants learned many new things and seemed to come away from the workshop with a renewed sense of purpose and commitment to the JML effort and the wish to schedule the next meeting of the JML community not only after another 3 years, but – maybe – already in 2014.