# A Report on the Current Semantic Framework of JML for Expression Evaluation and Undefinedness

Amritam Sarcar
Department of Computer Science
University of Texas at El Paso
El Paso, TX - 79968, USA
asarcar@miners.utep.edu

**Abstract**

Java Modeling Language (JML) has been around for more than a decade. It is one of the most popular BISL language tailored for Java. There are tools like Runtime Assertion Checker (RAC), Extended Static Checker (ESC/Java) and Full Static Program Verifier (FSPV) to support JML. RAC and ESC are the most popular ones since they have a low learning curve and are more catered towards developers [7]. RAC (which is also known as jmlc) is a compilation-based approach [10] wherein assertions are often built-into executable code. The translation scheme of these assertions into programming constructs is the focus of this paper. In JML2, the semantics for expression evaluation was local contextual-interpretation [11] which has now been recently changed to strong-validity [8]. This paper discusses the motivation behind the previous approach and the reasons behind JML adhering to the recent approach [17]. It also looks into the current semantic framework for translation of expressions and compares this to the previous approach. The author also compares and contrasts between the old, the current and the proposed framework for Quantified Expression translation, and proposes the best translation scheme.

## I. INTRODUCTION

The Java Modeling Language (JML) is a formal behavioral interface specification language that are not only used for documentation purpose but also to specify the behaviors of program modules in Java. Unlike Java, JML assertions are not limited by Java's expressions; JML introduces a varied set of constructs

which includes in-line assertions, quantifiers, invariants, history constriants, model programs and many more.

The nice thing about JML specification style is that they are written inside multi-line comments like /*@ ... @*/ or single-line comments like //@ ... The only difference is that for a Java compiler, the specifications are treated as comments, whereas for a JML compiler (like RAC) these comment-like specifications are translated into executable code [1].

## II. Background

### A. Expression Evaluation

JML is a superset of Java. However, unlike other specification languages, JML is built around Java. That is, on top of the existing keywords and special symbols of Java, JML introduces some of their own. The reason to introduce new language construct is to facilitate more stringent assertion checking.

### B. Undefinedness

Runtime Assertion Checker is essentially a tool that evaluates each expression at runtime and check with its corresponding assertion for violation. Since RAC evaulates at runtime, several situations may arise during evaluation. During execution of any code, exceptions may be thrown. In such cases what value is propagated to the client is a problem tha is associated with undefinedness.

### C. Non-executable expressions

JML is not only an executable specification language but also is used as a documentation language. For this purpose, JML introduced non-executable expressions like informal predicates (eg. **(* ... *)**). When a certain assertion cannot be formally defined, informal description is used. In general terms, informal expressions are not executable.

## III. Related Approaches

There has always been a constant effort to bridge the gap between logicians and software practicioners. Work on correctness of programs started in late 1960's. A common methodology - Leibniz's law - a deductive knowledge whereby new propositions, rules, axioms can be formulated was used a lot to infer program correctness.

$$\textbf{Leibniz} : P = Q \rightarrow E[v := P] = E[v := Q]; \tag{1}$$

[1]For further information, refer to [3], [17]

Equation 1 says that if the value of P equals to the value of Q, then any expression **E** which contains only P as a variable (and, some constant) is equal to any other expression **E** containing Q as any other variable. This expression is true only in the case where classical two-valued logic is followed. Conceptually, any state at runtime can be either true or false with respect to some attributes under some conditions. However, with the advent of new engineering constructs, different programming tools, it soon became evident that this may not be true. A classical example is the division operation. It is defined as -

$$y \neq 0 \Rightarrow y/y = 1 \tag{2}$$

because y is only defined in the range

$$\Re \times \Re - \{0\} \rightarrow \Re \tag{3}$$

However for the case in equation 1, if the expression E is 1/x, then even though x = y = 0, can we conclude 1/x = 1/y?

For the treatment of undefined expressions and partial functions, a third value $\perp$ (to represent undefined values), is assumed. In the works carried out by Bijlsma [2], the author proposes that an expression **E** is to be evaluated to *true* or *false* only if no subexpression yields an undefined state. However, in [13], it is shown that Bijlsma's work is not compositional.

There were yet other approaches like using abortive approaches to avoid undefinedness. Jones and Cheng [9] outlined two approaches. In the first approach, they proposed to transform partial functions to total functions by defining the domains appropriately (like in equation 2 for division). The problem with this approach is that it is not possible to implement this approach using a typed-programming language (like C/C++, Java etc.). Since every domain is binded to a particular type at compile-time, changing this domain at runtime and inferring any property is not possible. In the second approach, they proposed to view every function as a mapping from one domain to the other. This approach at times may not also be realisable.

Scott in his paper [18], suggests to assume all subexpressions (or, expressions) which evaluate to undefine as *false*. However, the law of trichotomy does not hold in this case. For example,

$$y = 0 \vee y < 0 \vee y > 0,$$

since for y = x/0 for some $x \neq 0$, all the sub-expressions have undefined terms. And, as per Scott, they evaluate to *false* which makes the entire expression to be *false*. However, since the LHS also contains undefined terms, it also evaluates to *false*, yielding *false* = *false*. This approach therefore is not implementable nor workable.

There were several proposals to extend the two-valued logic. Jones and Middelburg [15] developed a new variation of logic LPF for partial functions. They proposed an operator to extend undefinedness and two kinds of equality (weak and strong). However in [13], Gries and Schneider feel that Jones and Middleburgs' approach is far too complicated to be used in programming language.

More recently in [13], a new approach called *Avoding undefinedness through underspecification* was proposed. In this approach, every operation and function are *total* operations. This approach is an extension of [12]. For example, in this approach what would be the value of x/0? Since all operations are assumed to be defined, the author leaves the value of x/0 as to be unspecified. This value can be any value in the real-number line. This approach seems to be the preferred approach, whereby programmers can still use two-valued logic. This approach is formal, rigorous and unambiguous.

There are currently some tools that support assertions in various programming languages. The Jass tool [1] is a precompiler that supports assertions using the Design by Contract concepts and uses them in Java. iContract [16] is the first tool that provided thorough support for explicit specifications for Java. It is a source-code pre-processor that injects instrumented code in the source code itself. Eiffel [14] is another such programming language which supports assertions in their language construct itself. Assertions play a central part in the Eiffel design methodology which facilitates in building reliable OO software.

## IV. Contextual Interpretation: JML2 approach

In the previous semantic framework of RAC (JML2), the expression evaluation was based on contextual interpretation [11]. In JML2, RAC uses a game-strategy for reporting assertion violation. The strategy adopted is to predict whether the violation reported is to be positive or negative.

This approach is called a local contextual interpretation. It is local because if undefinedness occurs, it is determined by the smallest boolean sub-expression. It is contextual because the value of the small sub-expression is determined by the relative position of the expression.

### *A. Expression Evaluation*

JML extends Javas' expression by including quantifiers, set comprehension notations, and other specification and assertion constructs. The meaning of JML expressions are almost similar to Java expressions. However, there are certain major semantic difference between them.

- Abrupt completion: In Java, an expressions can either complete normally or abruptly, by throwing an exception. In Java, the value associated with this exception is not known, however it has an assoiciated reason (i.e., by the throw clause). Since JML is an extension to [13], it transforms

underspecified functions by always associating an arbitary value to an undefined sub-expression (see section IV-B).

- Evaluation order: Java has certain order-sensitive operators like the short-circuit operators such as && and ||. The right-hand side is only evaluated if any sub-expressions on the left-hand side does not yield a conclusive boolean value. The interesting part is when one of the sub-expressions complete abruptly, that is when there is a difference between JML and Java behavior. Table I shows that the standard rules of logic are followed even in presence of undefinedness.

| **Example** | **Value** | **Explanation** |
|---|---|---|
| x.length $> 0$ \|\| true | True or NullPointerException in Java | True, if x!=null, else the exception is thrown |
| x.length $> 0$ \|\| true | Always True in JML | Jf x==null then JML substitutes an arbitary value to x.length which evaluates to either true or false; however the right-hand side of the expression makes the whole expression true |

TABLE I

JML VS JAVA EXPRESSION EVALUATION IN PRESENCE OF EXCEPTION

- Executability: In JML, not all expressions are executable. Some expressions for eg. the informal descriptions and some quantifiers are not executable.

### *B. Semantics for Undefinedness*

The core of JML tools are built around the works of [13]. It is based on the classical two-valued logic where every expression can either have *true* or *false*. However, the RAC tool which is essentially a compiler that translates Java source code annotated with JML annotations into executable code. In JML [17], undefinedness can also occur due to unexecutable constructs (like (* informal description *)). To cope with this problem, JML extends the classical two-valued logic by introducing two different kind of undefinedness.

### *C. Angelic and Demonic Exceptions*

To adher to the two-valued logic, the developers of RAC decided to use two types of exceptions namely, Angelic and Demonic Exceptions. Demonic exceptions are those where runtime exceptions occur and is generally viewed as potential violation of assertion. Whereas, angelic expressions are not viewed as error conditions for which assertion violation should be reported. For demonic exceptions, the goal is to falsify the assertion under the rules of logic; for angelic exceptions, the goal is to make them true.

### D. The Implementation Framework

To support the different kinds of exceptions, there should be well-defined translation rules in place. Here, we discuss some of them in light of the semantic framework (More details can be found in [11]). JML has several constructs like field, class, interface, methods, and expressions. As discussed earlier, the formal specification written in JML annotation format is generally written abstractly without the worry of any computational faults like exception, runtime errors. JML chooses the best optimal strategy that would falsify the *top-level* assertion. The implementation framework and its associated translation rules should detect as many assertion violation with no false positives. The JML top-level assertions can either be associated with field, type declarations (concrete classes, interfaces, abstract classes, local classes, annonymous and others), methods or expressions.

The interesting part of the translation rule (see Table III) is the try-catch block. The behavior of this try-catch block is that during the translation of expression **E**, if the expression cannot be executed then it is caught in the catch block. In this block, the value of the expression is assigned a boolean value **!b**. The value of **b** signifies whether to falsify the top-level expression or not. This is to differentiate between *(*informal*)* and *!(*informal*)* and other such variants.

### E. Quantified Expressions

JML adds several new constructs like quantifiers. It contains several forms of them. An extensible and maintenable framework is adopted in the JML2 approach which provides multiple techniques to evaluate them. However, the most important evaluation strategy is to statically check for executability of quantifiers. It uses pattern matching to decide whether an expression is executable or not. An expression of the form *(∀ Object x; x!=null; x.contains(new String()));* is not executable since it is impossible to generate all the instances of an Object type. (This is not even workable since Object is the super-class of all other types). The general form of a quantifier is

$$(QuantifierV; P; Q);$$

### F. Discussion

The simplest approach for undefinedness in the runtime assertion checking point-of-view, is propagating to the user all exceptions thrown during evaluation. This approach is commonly practised in Java programming language as well as in [16], [14] and [1]. JML does not adher to this scheme because JML conforms to the standard rule of logic. Another reason, JML adopts this approach is because several

theorem provers like PVS uses two-valued logic with underspecifications. In fact, JML2 is the only Design-By-Contract tool that implements underspecified functions in runtime assertion checking tools.

## V. RAC Through Strong Validity

The JML2 approach outlined above had several problems. They are discussed in the following sections.

### A. Approximation of two-valued classical logic: Its shortcomings

Essentially JML2 approach approximated the classical two-valued logic with underspecification. However, in doing so, it gave birth to some anomalies. For example, let *x* and *y* be two array reference types that are null, then RAC interprets *x[0] == y[0]* as false and also *x[0] == y[1]* as false. However classical logic in theorem provers would evaluate the first expression to be as false and the latter as true. As per the works of [4], RAC does not implement the assertions based on classical logic.

Another shortcoming of the contextual interpretation approach is the loss of referential transparency [4]. Due to adhering to this approach, the implementation framework was also not optimised. In fact, due to each expression being translated into sequence of statements, the resulting instrumented code was so huge that there were cases [8] when the compiled code exceeded the maximum capacity, terminating the entire process.

### B. Motivation behind Strong Validity

The several problems resulting from the JML2 approach, lead the JML community to work collectively for a better semantic framework. This was primarily lead by Chalins' group which culminated in [6], [5], [4] and [8]; all are possible improvements from the previous approach. Runtime Assertion Checking tools are generally used by mainstream developers [5]. Thus, to cater to this need, the tools should be designed to aid/supplement formal verification.

*1) Assertions in Industry:* The survey conducted by Chalins'group [5] showed that practitioners, software engineers use assertions in their daily production code. The survey also shows how the practitioners want their assertion violation to work. A sample response to a question is shown in Table II. For complete results, check [5].

The survey result indicates that the industry is in favor of *reporting of errors/exceptions when a partial function is evaluated in RAC/ESC tools*. Some of the findings from this survey are -

1) Classical Logic: Previously, many software engineers argued on the fact that adoption of two-valued logic is adequate for program verification. However, this survey shows a different result. The laws

| Question | true | false | error | others |
|---|---|---|---|---|
| **nullReference** $> 0$ \|\| true | 6% | 5% | 84% | 4% |
| true \|\| **nullReference** $> 0$ | 73% | 1% | 18% | 7% |

TABLE II

RESPONSES TO QUESTIONS IN A SURVEY

of classical two-valued logic is not applicable to programming languages. For example, *1/x == 1/x* should not be interpreted as *true* for x = 0. This is the approach adopted by mainstream developers, because the programmer might have assumed that x would never be 0 in this context resulting to a potential error.

2) Partial functions: It is common practise where if illegal arguments are supplied then exceptions occur. To catch such exceptions guarded specifications should be used appropriately. Throwing of exceptions help detect more errors.
3) Ignoring Exceptions: If exceptions are ignored then possible origin of error cannot be located.
4) Consistency: ESC tools are static checkers which check for any possible assertion violation at compile-time. RAC tools are the ones that check for assertion violation at runtime. Both tools should be consistent in their behavior.

### C. Strong Validity in ESC/JAVA2

When the JML community agreed on the adoptation of strong validity [17], the first JML tool that adopted this approach was the ESC/JAVA tool. With the adoptation of this approach, more assertion violation was exposed. More than 50 errors were found with an increase of only 2% in processing time [6]. Due to consistency between RAC and ESC tools, it was obvious that RAC tools would also be re-implemented to conform strong validity.

### D. JML Runtime Assertion Checker (RAC)

To conform to the new approach, the implementation framework was re-designed. In [8], Chalin and Frederick created a new general expression translator. There was no more need for explicitly checking sub-expressions (see Table III). At runtime, the expression is evaluated in a top-level try-catch block that is used to catch two things: (a) JMLNonExecutableExceptions, and (b) all other exceptions. This approach simplifies expression evaluation but also increases the effectiveness of assertion reporting. From

the performance point-of-view, the instrumented code is reduced by 70% and the compilation time by 8%.

### *E. Translating Quantified Expressions*

Evaluation of Quantified expressions cannot be mechanically derived. The instrumented code is derived using static analysis. The new approach reuses the existing quantifier evaluation rules while wrapping them inside a local class scope. Unlike the previous approach, the local class approach embeds the instrumented code in the assertion position itself. The relative advantage over the previous approach is tabulated in Table IV.

## VI. A New Approach to Quantified Expression: Discussion

Quantified expression is one of the advanced features of JML. Writing a specification in a real-world scenario often makes use of several quantified expressions. Translating these quantified expressions into executable java code is the work of Runtime Assertion Checker. Since usage of quantifiers in JML specifications is common, there is a need to properly translate the quantified expressions into proper readable code.

Due to several problems found in the previous two approaches to handle quantified expressions, the author, at first compares the several alternative approaches and finally proposes one of the approach to be the optimal one. A total of five approaches were compared against a number of criterias. A brief explaination of the approaches follows.

### *A. Local-class Approach*

This approach like in JML2 [8] [17] is an approach where the translated code is embedded within a local class. For a detailed discussion see V-E.

### *B. Top-level Single-class Approach*

This approach is a variation of the recent JML2 approach [8], where only a single class is created. The nested quantifiers are translated inside the `while` loop. And, for multiple quantifiers different methods are created viz. `eval0(...)`, `eval1(...)` ... `evalN(...)` inside the same class. The only difference between the Local-class approach and this is that in the former for each quantifier, there is one class. In this case, for all quantifiers only one class is present.

| Local-Contextual | Strong Validity | Inline Approach |
|---|---|---|
| `try{`<br>` boolean racv0 = false;`<br>` /* translation of first quantifier */`<br>` racv0 = QE1;`<br>` if (!racv0) {`<br>`  /* translation of other quantifiers */`<br>`  racv0 &= QE1;`<br>` }`<br>` ...`<br>`}catch(JMLNonExecutable jmle0) {`<br>` racv0 = true;`<br>`}catch(Exception jmle1) {`<br>` racv0 = true;`<br>`}` | `try{`<br>` class racv{public boolean eval(){`<br>` /* translation of Quantifier 1 */`<br>` /* if Q has another quantifier, another local class is created here */`<br>` }`<br>` return ...;`<br>`}}`<br>` racv racvEval = new racv();`<br>` var = /* translation of other expressions */ && racvEval.eval()`<br>` && ... ;`<br>`}catch(JMLNonExecutable exception){`<br>` ... }`<br>`catch(Throwable v){`<br>` throw new JMLEvaulationError(v); }` | `try{`<br>` racV = Expression1 &&`<br>`  Expression2;`<br>` if (racV) {`<br>`  /* evaluate next quantified Expression ... */`<br>`  racV = /*some boolean value*/`<br>`  if (racV) {`<br>`  /* Similarly for other quantifiers we have try-catch blocks*/`<br>`  }`<br>`  }`<br>`  ...`<br>`}catch(JMLNonExecutable exception){ ... }`<br>`catch(Throwable v){ ... }` |

TABLE III

INSTRUMENTED WRAPPER CODE FOR THE THREE APPROACHES.

### *C. Method Approach*

In this approach the translated code of quantified expressions is embedded inside separate methods. No local classes are created. The same visitor classes are used for generation of instrumented code, only this time the wrapping is done around seperate methods and not in classes.

### *D. Statement Approach*

In this approach the translated code of quantified expressions is embedded at the statement level itself and bounded by try-catch blocks. No local classes are created for this purpose. For the statement level, two variations are proposed.

*1) Greedy-Analysis Approach:* In this approach, the quantified expression is evaluated first. The evaluated value of every quantified expression is stored in a data-structure. During evaluation if undefinedness occurs, then the exception is caught and is also stored. For this approach a seperate class/interface is created which returns either a boolean value or an exception as required.

*2) Inline Approach:* In this approach, the quantified expression is evaluated in place. That is, for every quantified expression, its corresponding instrumented code is generated in place. This simplifies

the quantified expression evaluation a lot. However, this requires a change in the expression evaluation strategy to accomodate the inline expression evaluation.

## VII. Evaluation criteria

Several evaluation metrics were taken into consideration for accessing the best framework to translate quantified expressions into Java executable code (RAC code).

### A. Readibility

The common JML tool a.k.a the JMLRac compiler has a *-P* option in which the translated code in Java source code format is displayed. This is to facilitate programmers, JML developers and users to check for correct implementation, and is also used for debugging purpose. Thus readibility plays a key role while translating quantifiers. If the corresponding Java source code for the quantified expression is distributed around, then it may hamper the natural flow of understanding by the reviewer.

### B. Supports all features of quantifiers

The framework that would ultimately be chosen should be able to support all the features of quantified expressions. The framework should support translation at every place where quantifiers can be used like in class, method or inline specifications. Ideally all types of quantified expressions should be supported by the frameowork. It should also be able to support nested quantified expressions (For example *(∀ T1 e1; P1 ; (∃ T1 e1; P2 ; Q2));*).

### C. Supports expression translation in all scenarios

The previous section (VII-B) illustrates that expressions can be associated with classes, inlines or method specifications. However, all these three specifications can occur in different scenarios. The framework should be able to translate the code accordingly. Some of the scenarios where quantified expressions can occur are, in interfaces, abstract classes, local class, static methods or static classes and even in annonymous classes. The different approaches should be able to translate the code in all of these scenarios.

### D. Translation problems

Another important criteria is the problem of implementing the desired framework. This problem can be categorised into two sub-problems. They are -

*1) Translation of quantifiers:* Translation of quantifers into basic sequence statements would be same across all the approaches. The basic framework for translation would be very similar to the JML2 approach [10] [8]. However, this should be done in a systematic manner such that the framework is extensible and should also not change the exisiting framework of the Eclipse compiler [7].

*2) Wrapping and value passing:* Each of these translated code need to be wrapped either into a method, class or as a sequence of statements with proper try-catch blocks. Since all of these approaches has different needs like in method approach all the parameters in the *checkMethod* are also propagated into the quantified translation method. This is same in the case for top-level-single-class approach where the method parameters are declared final (by, default) so that the local class can access them. Since determining which of the variables are to pass into the function or local class, a visitor method is required to check which variables to be used. This may however complicate the implementation.

### *E. Performance*

Performance in this context may come into two flavors. The static compilation time may be effected by the framework that is chosen. However, since all the approaches have the same basic framwork (only the wrapping is different) they all have very similar compilation time[2]. However this is not the case for the runtime performance. Even though an explicit study has not been done, but based on the initial results (see Figure 2), the local-class approach( [8]) is the slowest. Figure 1 shows how the bytecode is also affected due to different approaches. It is also a fact that the time to load the class, create an object and call the appropriate method with the right parameters takes up much time than executing similar length of sequence of straight-line code. Figures 1, 2 both show comparitive performance wrt to Inline approach. They are calculated by the follwing formula:-

$$P_i = (\frac{A_i}{Baseline} - Baseline) \times 100;\ where\, Baseline\ =\ Best\, approach. \tag{4}$$

In our case it is observed that in both cases, Inline approach is the Best approach.

### *F. Implemented framework should be extensible and maintainable*

The framework should cater to the goals of JML4 approach ( [7]). It should not create unnecessary extensible points and make future implementation difficult. The implementation framework should be scalable and extensible. It should also be maintainable so that it is easy to debug and make necessary changes to the implementation framework as needed.

[2]Strictly, there is a very high probability that the local-class approach would be the slowest since it requires to create several other local classes (for every quantified expressions)

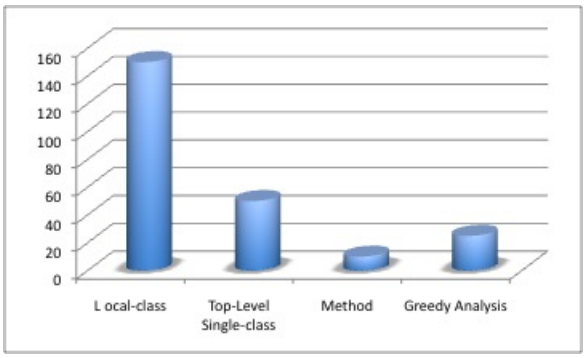


Fig. 1. Different bytecode size: This Figure shows by how much % are the different approaches larger in size compared to Inline approach.

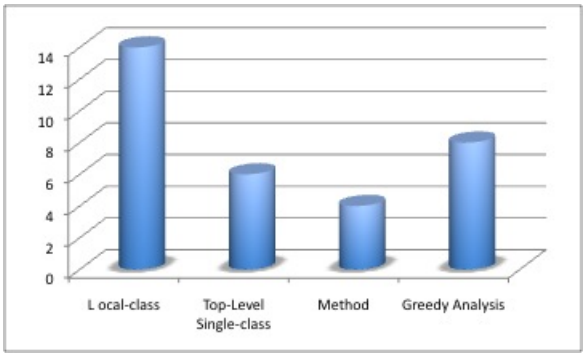


Fig. 2. Different Runtime speed: : This Figure shows by how much % are the different approaches slower compared to Inline approach.

## VIII. Evaluation

Table IV tabulates the relative differences between all the approaches across different evaluation criteria. It is evident from the table that, **Inline Approach** seems to be the best.

## IX. Conclusions

The JML2 approach (Cheon and Leavens), was based on local contextual interpretation. The motivation behind this was to conform to two-valued logic and also so that ESC and RAC tools behave similarly. However, in the works followed by Chalin, he showed that proper semantic meaning should be designed keeping the practitioners in mind. Hence, he proposed a new semantics which was based on Strong Validity. With changes in semantic meaning, there was also an implementation framework change in quantified expression translation. From statement translation to local-class, there were many translation

| Approaches | Readibilty | Support All features of Q.E. | Support translation in all scenarios | Translation problems | Performance wrt Inline | Extensible and Maintenable |
|---|---|---|---|---|---|---|
| Local-Class | Embeds code inside local class. It's fairly readable. | Does not support in-line assertions | Supports translation in all scenario except when assertions are inside annonymous class | Already implemented | Slower by 14% and larger by 150% | It is extensible and maintenable |
| Top-level-Single-class | Embeds code inside a single class in diff. method. It's more readable | Does not support in-line assertions Is implemented after code restructuring | Supports translation in all scenario except when assertions are inside annonymous class | Can easily be implemented | Slower by 6% and larger by 50% | It is extensible and maintenable |
| Method | Embeds code inside diff. methods. Code is distributed | Supports all features. | Supports translation in all scenario. May cause a problem for annonymous class | Can easily be implemented. Wrapping is different | Slower by 4% and larger by 10% | It is extensible and maintenable |
| Greedy-Analysis | Embeds code inside separate try-catch block. There are many try-catch blocks | Supports all features. | Supports translation in all scenario | Can be implemented. May be a problem for creating interface | Slower by 8% and larger by 25% | It is extensible and maintenable |
| Inline | Embeds code inside separate try-catch block. It is inline | Supports all features. | Supports translation in all scenario | Can be implemented. Problem in refactoring expression translation | | Due to expression translation framework is not easily extensible |

TABLE IV

EVALUATION OF DIFFERENT APPROACHES ACROSS DIFFERENT

techniques. However the comparitive study conducted in this paper shows, adhering to Inline Approach is the best strategy available.

## X. Acknowledgements

The author would like to thank the following people for their insightful comments on earlier drafts of this paper: D. Novick and Y.Cheon.

This effort was part of the coursework that the author took during his graduate study at University of Texas at El Paso.

## References

[1] Bartetzko, D., Fischer, C., Moller, M., and Wehrheim, H. Jass - java with assertions. *Electronic Notes in Theoretical Computer Science 55*, 2 (2001), 1–15.

[2] Bijlsma, A. Semantics of quasi-boolean expressions. *Beauty is our business: a birthday salute to Edsger W. Dijkstra* (1990), 27–35.

[3] Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G., Leino, K., and Poll, E. An overview of jml tools and applications. *International Journal on Software Tools for Technology Transfer (STTT) 7*, 3 (June 2005), 212–232.

[4] Chalin, P. Reassessing jml's logical foundation. In *Proceedings of the 7th Workshop on Formal Techniques for Java-like Programs (FTfJP'05)* (2005).

[5] Chalin, P. Are the logical foundations of verifying compiler prototypes matching user expectations? *Form. Asp. Comput. 19*, 2 (2007), 139–158.

[6] Chalin, P. A sound assertion semantics for the dependable systems evolution verifying compiler. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 23–33.

[7] Chalin, P., James, P. R., and Karabotsos, G. An integrated verification environment for jml: Architecture and early results. In *Sixth International Workshop on Specification and Verification of Component-Based Systems* (September 2007), ACM Press.

[8] Chalin, P., and Rioux, F. Jml runtime assertion checking: Improved error reporting and efficiency using strong validity. In *FM '08: Proceedings of the 15th international symposium on Formal Methods* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 246–261.

[9] Cheng, J., and Jones, C. On the usability of logics which handle partial functions. *In: Morgan, C., Woodcock, J.C.P. (Eds.), 3rd Refinement Workshop* (1991), 51–69.

[10] Cheon, Y., and Leavens, G. T. A runtime assertion checker for the java modeling language (jml). In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP 02), Las Vegas* (2002), CSREA Press.

[11] Cheon, Y., and Leavens, G. T. A contextual interpretation of undefinedness for runtime assertion checking. In *AADEBUG'05: Proceedings of the sixth international symposium on Automated analysis-driven debugging* (New York, NY, USA, 2005), ACM, pp. 149–158.

[12] Constable, R. L., and O'Donnell, M. J. *A Programming Logic* (1978).

[13] Gries, D., and Schneider, F. B. Avoiding the undefined by underspecification. Tech. rep., Ithaca, NY, USA, 1995.

[14] Howard, R. The eiffel programming language. *Dr. Dobb's J. 18*, 11 (1993), 68–73.

[15] Jones, C. B., and Middelburg, C. A. A typed logic of partial functions reconstructed classically. *Acta Informatica 31* (1994), 399–430.

[16] KRAMER, R. icontract-the java design by contract tool. In *Proceedings of Technology of Object-Oriented Languages* (1998), pp. 295–307.

[17] LEAVENS, G. T., POLL, E., CLIFTON, C., CHEON, Y., RUBY, C., COK, D., AND KINIRY, J. Jml reference manual draft, $revision$ : 1.68, 2002–08.

[18] SCOTT, D. Existence and description in formal logic. *In Ralph Schoenman (ed.) Bertrand Russell, Philosopher of the Century* (1967), 181–200.