Improving Java Verification of Quantifier Comprehensions in OpenJML

Group 22

Colin Herzberg Robert Derienzo Robinson Vasquez
Sachin Shah

Sponsored by Dr. Gary Leavens

Department of Computer Science
University of Central Florida
Fall 2021

Contents

1 Executive Summary

When code is written bugs will appear. With many critical systems operating on computers, we need to be able to confirm that code will work. Testing is a major component of software design. There are many layers of testing from unit testing individual methods to user acceptance tests on entire systems. With all of these testing methodologies and frameworks we still have bugs.

OpenJML is a framework to allow formal verification of program correctness to be conducted on software. This means you can write your Java code with OpenJML annotations, and prove whether or not your program will work on any possible (allowed) input. Knowing that code will work before we deploy it is key when there is no chance to go back and update it, or where a single bug could mean a loss of life. The first critical step in development on the OpenJML project was not development at all, but extensive planning and research. We began with assembling a roadmap and plan of action to implement the three major quantifier comprehensions in OpenJML. These quantifier comprehensions find an output when given an expression, variable, and a range. For example, the \sum quantifier when given an array and a range of one to five would return the sum of the values within that array with index between one and five. Originally our project was to implement five quantifiers but upon discussion with our sponsor and his associate Dr. Cok, we have come to realize two of the quantifiers have been implemented already.After this discussion with our sponsor, our group shifted scope and focused on the implementation of three quantifiers total including: \sum, \num_of, and \product.

After a base understanding was established, our group developed test-cases in Java annotated with OpenJML in accordance to our test driven development plan. Once we had created these, we set out to develop an effective method of translating them into SMT-LIB. A separate language used by SMT solvers to verify whether or not the software is satisfiable or not. In conducting this translation we constructed a plan on how to axiomatize these quantifiers. With this plan and a good test bed, we began to implement these features within the extended static checker of OpenJML. Pending success in static checking, we will also work to improve the quantifier comprehension features within the run-time assertion checker.

This project was focused on research and investigation in order to understand formal verification. We have conducted significant research into topics such as SMT solving, design by contract, specification languages, and formal verification methods. By conducting significant research in Senior Design I we have prepared ourselves to implement in Senior Design II.

2 Project Background

2.1 Behavioral Interface Specification Languages

A behavioral interface specification language is a framework for creating annotations using a predefined syntax. This language is used to note what the code around it is doing, using comments that are formally defined rather than using written language. With the knowledge that a programming language is a formally defined syntax that describes commands that a computer can execute, we can recognize that a behavioral interface specification language might be able to define commands for a computer. [1]

Using these annotations to analyze program correctness extends the usefulness of using a syntactically complex behavioral interface specification language. These languages are supposed to reduce development time by preventing bugs and confusion about use of code. The value of writing it, must outweigh the time used to write the annotations.

A formally defined specification language has more benefits than just being able to be read by a computer. When programmers from around the world code, they use the same syntax to describe what their software will do. Consider for example, ordering chips at a restaurant. In the United States you might end up with a paper thin crispy potato, but in England you will get a long rectangular extrusion of potato. When describing code using culturally specific syntax and lexicon there can be significant ambiguity leading to confusion. With the formally defined syntax, code can be shared around the world and anyone that reads the comments can decipher exactly what it means just by reading the syntax specifications.

Specification languages describe specific features of the programming languages that they are specifying. For example, a specification language for Java, an object oriented, procedural programming language, might require different annotations than a specification language describing a functional language such as LISP. Since OpenJML is a specification language for Java we will focus on specifications for Java programs.

Some of the most common bugs within Java development are when invalid inputs are passed to method parameters, and when invalid or unexpected outputs return from a function. For example, you might have a function as follows

```
public static int sqrRoot(int n) {
    int result = 0;
    while(result*result != n)
```

```
        result++;
    return result;
}
```

When this program gets an input that is not a positive integer it will lead to an infinite loop. Java does have us covered to some extent by requiring a type to be specified on variables. When compiling a programming that passes a number that is not an integer into this method, compiler will throw an error. The problem is, if you pass a negative integer into this method, there is nothing to catch the problem.

One way you might go about dealing with this problem is using a formal specification so that other developers using this method know that the inputs must be positive. This might look like the following.

```
//@ requires n >= 0;
public static int sqrRoot(int n) {
    int result = 0;
    while(result*result != n)
        result++;
    return result;
}
```

This specification in the JML syntax is formally defined. With this, we can use an automated system to check if n satisfies the condition proposed. Similarly, we can also verify that function outputs fit a specification. We can use this methodology to create syntax that describes the integral features of our programming language.

In order to increase adoption of behavioral interface specification languages they must be intuitive to write. Given this, it would be beneficial to have specifications follow a syntax that programmers might already be more familiar with. When writing a program we tend to use built in methods to increase efficiency and reduce redundancy. Similarly, we can add specifications that have a formally defined meaning to our specification languages. The five quantifiers that we are working with are commonly used in blocks of code in Java. By making the syntax more concise, we can reduce development time, and also reduce the amount of possible bugs in the specification [1].

Yes, specifications can have bugs. What the developer writes might not mean what they

thought. For example:

```
//@ ensures \result <= 0;
public static int F(int n) {
    return n * n;
}
```

The above program simply squares the given integer. As we know, every squared integer will be positive. The above specification notes that the output of the method must be less than zero. The problem is we can only assume that the method is correct, and the same about the specification. At this point, we can visually inspect and know that there is a problem. We can also check with an automated tool which would provide an error. Both of the following methods would correct the issue, but they now have entirely different outputs.

```
//@ ensures \result >= 0;
public static int F(int n) {
    return n * n;
}

//@ ensures \result <= 0;
public static int G(int n) {
    return (n * n) * (-1);
}
```

Clearly, as methods become more complex this problem can become even more of an issue. At the same time, many would argue that a false positive is much better than no warning at all. Given that we have opportunity to recognize that the program might not do what the specification describes we are less prone to bugs.

2.2 Program Verification and Design by Contract

Many software engineers use test driven development to ensure the validity of software. Unit, integration, and functional tests are just a few types utilized by most teams. A critical question to ask is “when has the software been sufficiently tested”. Stopping too early can lead to system failures and bugs. While stopping too late can lead to missing deadlines. Because testing is human driven, there is always an uncertainty that an edge case was missed.

One mechanism to be more confident in software validity is to use contract programming. This method involves defining precise and verifiable specifications - or "contracts" - for every component. Design by contract was first introduced by Bertrand Meyer with the Eiffel programming language. This style of programming means component programmers must fulfill the contract and component users must satisfy the conditions of use. Three primary questions should be answered by the contract:

1. What does the function expect?
2. What does the function guarantee?
3. What does the function maintain?

The benefit is software engineers can better know what to expect from using any component. In dynamic languages such as Python this is specifically beneficial as there is not compile time type enforcement. Many languages include design by contract support natively such as Eiffel, Scala, Kotlin, D, and Clojure.

Some languages include "external" support. These methods either add design by contract functionality by introducing special functions or adding syntax in comments that can be parsed by a third-party tool (like OpenJML). Some examples are C, Go, Rust, JavaScript and Python.

Python is specifically an interesting case. Projects like 'mypy' that do static type checking of Python programs demonstrate the need for program verification. PEP 316 introduces programming by contracts to Python. Although it is currently deferred, it presents interest concepts for how to write contracts. It's designed to be an opt-in and is inspired by Eiffel. This PEP has led to Python modules that augment Python to allow for runtime contract checks such as the Contracts package. This project is interesting as Python is a dynamic language that can be harder to prove statements about than in Java.

C# is another import example as the Spec# or CodeContracts library is most similar to the OpenJML project. In fact, many research papers about program verification use Spec#. This is primarily a research tool and not a production instrument.

Contracts can be specified in two major ways: code and comments. Writing contracts as code involves adding asserts for pre and post conditions. At run time, the software will confirm the conditions are met. This may cause a small performance hit so may be compiled away for production use cases.

Writing contracts as comments adds specifications in commented blocks (such as OpenJML). These are not utilized at run time by default, and instead are used to statically check the program. This allows contract syntax to be separated from the core language. This is important as contracts should generally not be language specific and rather focus on the underlying logic.

These contracts can be used to statically verify program correctness. By analyzing component implementation and contract specification, it would be possible to formally verify if the contract is fulfilled. Then, as long as each component user follows the contract conditions, the software would be verifiably correct.

2.3 Java Modeling Language and OpenJML

2.3.1 What is JML and How Does JML Specify Programs

JML is a Behavioral interface specification language as described in the previous section. Designed by Gary Leavens and his teams, JML was created to research how a specification language can be designed so that it can be used in a production software environment. Given that formal program specification and verification are still quite new and not broadly accepted within enterprise software, JML hopes to make software verification more approachable and useful. [2] By creating a language that emulates the syntax of Java, JML should not be a challenge for a Java programmer to use.

Eventually, through collaborations with Rustan Leino and Greg Nelson, as well as others, JML was used to define static verification as well as run-time assertion checking of Java programs. Later on, David Cok brought JML tools to openJDK helping to establish OpenJML, the open-source JML project. [2]

OpenJML provides users with two options for formal verification of program correctness. First of which is run-time assertion checking which focuses on providing a quick helpful check for developers who care about correctness, but do not want to invest significant time testing every case possible. On the other hand, by using the extended static checker, users can be confident that the program will react as expected given any inputs that meet the criteria specified.

With the goals of OpenJML including ease of use for developers, adding features to improve development time is of utmost importance. While it is possible to verify some of the quanti-

fiers, in a more simple fashion, others would take some complex logic. By adding quantifier comprehensions, we can significantly decrease the effort to verify a broad range of programs. As we have seen in the quantifier sections, there are many use cases for each quantifier. By adding these features, we hope to increase the adoption of OpenJML.

2.3.2 Who is OpenJML For?

OpenJML can be used by anyone who creates Java-based software. There are two primary types of users.

1. 'Learner': Someone who is learning about formal logic and best software practices. For this type of user, it is important that JML statements are easy to use and straight forward. Simple functions should be equally simple to verify.

2. 'Verifier': Someone who is in a critical role in a software team. This user cares dramatically about the correctness of the software meaning the JML statements must be well defined and correct. Catching bugs is the primary concern of this user so error reports should be complete and informative.

All users will care about the speed of verification, as faster proving will allow the user to use the tool more passively throughout development. OpenJML also implicitly serves as 'contract' based programming. The runtime assertion checker confirms each functions 'contract' is being upheld so any pre/post conditions should pass.

OpenJML should be viewed not only as a testing tool but also as additional documentation. By reading JML annotations a software user can fully understand what to expect from using that function.

Amazon is the most well known entity that uses OpenJML verification [3]. Amazon's mission is to have security assurance that is backed by mathematical proofs. The AWS Encryption SDK for Java uses OpenJML's static checker to verify the package. In fact, Amazon funds provable security research such as "Practical Methods for Reasoning About Java 8's Functional Programming Features" which discusses functional programming verification in OpenJML [4].

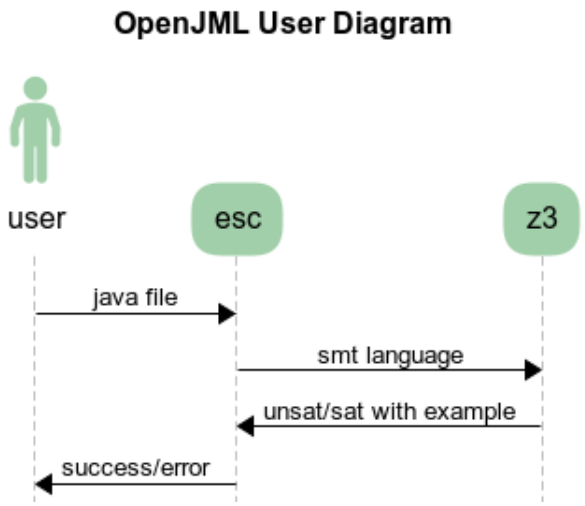


Figure 1: JML User Diagram

Using OpenJML is quite easy. After annotating a Java class with JML statements, one can run the extended static checker with 'java -jar openjml.jar -esc CLASS' or the runtime assertion checker with 'java -jar openjml -rac CLASS' (Figure 1). As of OpenJML 0.16.0, the tool packages JVM 16 in the executable allowing the user directly run OpenJML without 'java -jar'.

2.4 Formal Logic

Many sorted first order logic comprises the formal system behind SMT-LIB. To build an understanding of this system we will go over regular classical first order logic, propositional logic and formal logics in general. We will also discuss Model theory, which is generally the theory that is used to formalize the semantics of terms and formulas in various logics.

Formal logic is a system consisting of an alphabet, a grammar that defines the well formed formulas with respect to the alphabet, inference rules that are used to derive well formed formulas from previous ones and axioms that are taken as true by the system. First order logic is a formal logic that is an extension of propositional logic.

Classically, propositional logic consist of the usual logical connectives that are taken by the system as predefined primitive symbols. How connectives affect the truth value of the formulas they are a part of can be defined through truth tables or inference rules. It also consists of propositional variables which stand for the formulas, it has a grammar that dictates which formulas are well formed. It can have various inference rules, such as Modus

Ponens, or Modus Tollens. Inference rules have the following syntax:

$$\begin{aligned} & p \\ & p \rightarrow q \\ \therefore\ & q \end{aligned}$$

Which means that one can derive q from the statements p and $p \rightarrow q$ The example above is called modus ponens.the semantics of propositional logic is given by set theory using valuation functions v that assign truth values to the atomic formulas. Atomic formulas are propositional variables with no connectives and they represent the smallest possible formulas in propositional logic. Valuation functions have a predefined output given certain connectives for example $v(a \wedge b) = True$ if and only if $v(a)$ and $v(b)$ are true and false otherwise. So logical connectives restrict the behavior of the valuation functions and valuation functions can be used to define logical connectives.

First order logic extends propositional logic by adding predicates, functions, constants and quantifiers.

The most detailed way to express an assertion of the form "Carl lives in the US" in propositional logic is by using propositional variables. In first order logic however, the propositional variables are replaced by predicates that take one or more constants as input allowing more detailed formalized assertions than the one above. One possible formalization of the above assertion is $Lives(Carl, US)$. Functions are a routine that take constants and output some other constant, where constants stand for subjects or objects in English sentences. The quantifiers are the existential quantifier and the universal quantifier. First order logic tries to more accurately map natural language arguments to a formal logic since natural language assigns predicates to objects and subjects. The quantifiers are binders for variables in well formed formulas in first order logic, their importance is most evident when semantics are provided to the system. Modus ponens can be expressed in the same way as in propositional logic but with the understanding that p and q are sentences in first order logic and sentences in first order logic take the following form:

- if p has the form $P(a_1, \ldots, a_n)$ then p is a sentence, any of the $a_1, \ldots, a_n$ can be the outputs of functions so for example a_n could be $f(c_n)$
- if p and q are sentences then $p \wedge q$, $p \vee q$, $p \rightarrow q$, $p \leftrightarrow q$ and $\neg p$ are all sentences
- if p is a sentence then $\exists xp$ and $\forall xp$ is a sentence where x could be free in p.

The inference rules of first order logic includes the ones from propositional logic and some more rules dealing with quantifiers like Universal instantation which says that

$$\forall x p$$
$$\therefore p[a/x]$$

$p[a/x]$ represents the sentence obtained by substituting every instance of a free x in p by a

The semantics of first order logic is defined using model theory. In model theory there are models or structures which give an interpretation to the predicate symbols, function symbols and constants. Based on that interpretation truth is defined in a model. Models consist of a set called the universe of the model and a function which interprets the predicate, function and constant symbols.

Models still interpret the logical connectives the same way that valuation functions do in propositional logic.

Valid sentences are sentences true in every model, contradictions are sentences false in every model. Satisfiable sentences are sentences true in some model. Sentences are formulas with no free variables

Many-sorted first order logic is a generalization of classical first order logic, in fact the first order logic just described can be called a 1-sorted first order logic, since all constants are of the same sort. Sorts are like types or objects that define the behavior of certain constant, certain functions on those constants and certain predicates on those constants. So for example you could be able to apply the summation function to constants of sort Int but not of sort Bool.

Many-sorted first order logic adds sorts to first-order logic. Every constant symbol is assigned a sort symbol that represents the sort the constant is a member of. Many-sorted first order logic has a set of sorts.

How this affects the model-theoretic view of semantics is that the Universe U of the Model M has subsets and the model interprets sort symbols as subsets of the universe U and constant symbols of sort σ are interpreted as constants which are members of the subset σ^M representing that the constant is part of that type or that sort. Furthermore, Predicates and functions are now type so the inputs to functions are typed and the output to functions are typed and the same way for predicates, the predicates are applied over constants of certain types.

Truth in many-sorted first order logic behaves similarly to truth in first order logic the only

difference is the behavior of functions and predicates. Logical connectives have the same meaning.

Many-sorted first order logic is very useful because in natural language we often apply types or categories to various objects and use predicates or exclude predicates and functions to constants based on their type.

This logic is also particularly useful for this project, because programming languages are typed, they have a type system which can be better modeled by many-sorted logic. The type systems define the behavior of the members of that type and how they interact with each other and what behaviors between the different types can be assigned. It is particularly useful for Java since Java is a strongly typed language.

2.5 Related Work

One of the most useful papers for the development of quantifier comprehension is a paper by K. Rustan M. Leino and Rosemary Monahan called "Reasoning about Comprehensions with First-Order SMT Solvers" [5].

The paper discusses translations of quantifier comprehension from the Spec# specifications program verifier to BoogiePL which is the language accepted by various SMT solvers used by the authors. This is of course relevant to our work because it is exactly what we are trying to do only with another specification language and translating to another language for the SMT. We are using OpenJML as our specification language and SMT-LIB as the language we are translating to, to use as input for the SMT solver z3.

The paper explains that one needs to develop an axiomatization for the quantifiers so that the solver has a syntactic understanding of of how the quantifiers work. It also goes over matching triggers that are used to instantiate quantifiers using a data structure called an e-graph. So matching triggers are expressions with free variables like g(x), the e-graph contains terms of the form g(5) or g(3) and by matching the trigger with the ground terms in the e-graph, quantifiers are instantiated with the members of the grounds terms, in this case 5 and 3. We, like the paper suggests, need to develop an encoding, an axiomatization, and matching triggers for our quantifiers.

3 Project Overview and Motivation

3.1 Function of the Project

The Java Modeling Language is a program specification that has been in active development since 1999. The specification does nothing itself but allows for the creation of tools to interpret it. The first example of one such tool is OpenJML. OpenJML is currently capable of checking Java programs that have been annotated with the JML language and supports many of JML's features. The broad goal of the OpenJML project is to create a fully capable tool for JML that is easy to use and robust enough to inspire use. OpenJML as a whole seeks to seeks to facilitate verification of Java software and as a result, increase the reliability and correctness of computer software. Currently the largest use of openJML is in academia but as it grows more robust and leaves less negative space in terms of JML's domain OpenJML hopes to be adopted by students, researchers and industry alike. OpenJML does not check or 'prove' programs itself, rather it translates the annotated JML expressions into SMT-LIB, which its back-end comprised of a SMT solver can handle.

This conversion process is handled in one of two ways, depending on how the user has specified their code to be verified.

1. Runtime assertion checking - Verification is performed at runtime and no conversion is done. Instead the code is padded with additional functions at compile time whose purpose is to verify the statements at runtime
2. Extended Static checking - Verification is performed in a static manner, where the code goes through multiple phases eventually being converted to SMT-LIB. From there it is passed into a SMT-Solver and the solvers return is used.

Our project is to expand the domain and usefulness of OpenJML by support for five additional quantifiers within the extended static checker, namely: max, min, number of, product and sum.

- Max – Maximum value of expression
- Min – Minimum value of expression
- Number Of – Number of elements

- Product – Value of expression × expression
- Sum – Value of expression + expression

The above are 5 generalized quantifiers that our product will introduce into OpenJML which return the maximum, minimum, number of elements, product, or sum of the values of the provided expressions, where the variables satisfy the given range respectively. By growing the domain that is covered by OpenJML through the addition of these 5 quantifiers, not only do we make it more robust; we potentially extend the use-cases of the project as a whole. Our project's addition to OpenJML is in line with the overall goals of OpenJML, and will expand the project as a whole.

3.2 Motivations and Importance

3.2.1 Motivations

Many sectors in society make use of software to attain various important objectives. We use software to efficiently distribute vaccines among the population, manage millions of customers' bank accounts, schedule classes for millions of students, and make models and develop predictions, among many other essential tasks. Given the relevance of software-based tools on the functioning of many aspects of society, it is imperative that as software developers we make sure that code works as intended. Some ways we use to attempt to make sure programs work correctly are:

1. Developing coding standards that software developers can adhere to when developing software which reduces the probability of making bugs
2. Reviewing code with the help of other software developers and experts.
3. Developing tests for possible edge cases one might have missed

Nevertheless, these methods do not assure us that the code works as intended. Errors in code can lead to very dire consequences, like slower vaccination rates, customer monetary loss, scheduling conflict for students, or inaccurate predictions and security risks.

One possible solution for this concern is the use of formal methods to prove program correctness. Formal methods would allow us to prove assertions about our programs given that

our assumptions about the program are true. Our project deals with OpenJML which is a behavioral interface specification language for Java. OpenJML seeks to allow users to check whether Java code meets certain specifications. In other words, the motivation behind OpenJML is to have a tool that allows one to systematically prove the correctness of certain programs in Java using formal methods.

By developing a method to prove programs we will be able to provide more security to customers that use our software. We will be able to save money for software companies through the reduction of errors that can cost from little to vast amounts of money. We will also be able to reduce harm to people that rely on various software applications. We will also be able to be more certain of our predictions and assessments using software tools.

3.2.2 Personal Interests

Colin Herzberg:

I have personally written my fair share of buggy code. Every developer has caused an error that has led to some downstream consequences. To help prevent errors we have many layers of testing from unit tests to user acceptance tests. Many develop code in a test-driven environment focusing on finding tough cases and verifying that their code works on the examples they come up with. The push for more automated testing is great but only as good as the tests that the user can create. If we forget about an edge case while testing, the program will fail. As we know, this can happen all too easily. OpenJML looks to solve some of the problems of verifying that software is going to do what we want it to. With OpenJML, we do not need to be looking for edge cases that could break our code, but instead allow the computer to help us. If we can formally prove our software works, there will be a lot fewer phone calls to on-call developers who were wishing to sleep through the night (I like my eight hours of sleep).

At the same time, my interest in expanding my education with classes including Systems Software and Computational Discrete Structures has motivated me to pursue this project. As we approached more complex problems such as the Halting Problem and Satisfiability, I found myself asking more questions. This project will give me an opportunity to understand some of the applications of the topics, and hopefully answer some of my questions in the process.

With much of today's software development being focused on the web, it's interesting to focus on a topic that is more theoretical. I find myself wanting to develop skills in understanding

academic and research-focused topics. With experience reading and understanding these topics, it will be easier to maintain an academic mindset throughout my life and stay on top of new, cutting-edge, technology.

Finally, I am happy to be contributing to an open-source project. The open-source movement allows for development teams from all over the world including big tech and college students to use similar tools to create with. You never know where the next great idea might come from. Allowing anyone the tools they need to realize their ideas creates a better world with more innovation.

Robert DeRienzo:

OpenJML is a suite of utilities designed to implement the JML specification language.

Overall, the goal of OpenJML is to create an easy-to-use tool that can specify and then verify Java programs. While doing program verification in this manner is used primarily in academia, OpenJML hopes to expand software verification into the industry.

It is important to write correct code. More so when the code will be used in critical infrastructure when people's lives could be at stake. It is then ironic how incredibly surreptitiously critical bugs/errors find themselves in production stage applications. This is why many other developers and I, place such a large focus on developing test cases. Test cases for this, that, and the other. OpenJML seeks to verify programs using logic to open the possibility of more robust testing with less development time.

Furthermore, JML and by relation openJML both provide useful tools to the research community. While the ultimate goal is to expand this focus in the future; it is worth mentioning that I support the initiative to make this particular subset of research more access able.

My general interest in this project stems from the enjoyment of Logic and Discrete Mathematics. Throughout my university classes, I have enjoyed Discrete I & II more than any of my other classes. Going further down the line I find enjoyment in puzzles and things of that nature. It is enjoyable to break down problems into simple components and then, using formal rules and theorems, construct a solution that is undeniably correct. This project in particular (being OpenJML) is used to specify and verify programs written in java which I find pretty closely related to my interests in both logic and Object-Oriented Programming as a whole.

Outside of this, and probably most importantly I chose this project because I believe it will help me grow as a Computer Scientist and learn new things.

Robinson Vasquez:

My personal interest in this project comes from my fascination with formal systems and particularly formal logic. This project uses many-sorted first-order logic to try and prove program correctness. In addition, it utilizes various concepts related to formal systems like SMT, satisfiability, axioms, inference rules, and formal theories. Through this project, I hope to expand my understanding of formal logic and reasoning with typed languages and to apply these methods to programming tools.

Moreover, the language to which we are translating to called the SMT-LIB language has a strong formal grounding. Through the study of this language, I will explore concepts like many-sorted first-order logic, sorts, predicates as functions, and constants are functions and model theory and its relationship to axioms since both are used to interpret function symbols. I want to have a concrete, rigorous and good understanding of all these concepts and I believe I will be able to greatly expand my understanding of these concepts through these projects. I also have been mostly exposed to classical one sorted logic, now it will be a great opportunity to learn many-sorted logic and apply it to program analysis.

Furthermore, I believe working on this project will help me when thinking informally about programs that I am developing so that I can catch mistakes easier, and realize quicker whenever some part of the code works as intended or not.

I also want to get a Ph.D. in something related to formal systems and formal methods and their use in computer science, and I believe working on this project will help me towards realizing that goal.

Sachin Shah:

Test-driven development is a powerful method to create robust software. However, satisfying passing all test cases can be; it is never enough to be 100% certain the software will perform as expected. Missing edge cases and ill-defined function requirements spell disaster for the wild west that is the real-world production environment. As software proliferates into increasing aspects of our lives, it is critical to be confident in our codes' robustness. OpenJML provides a method rooted in formal logic to do just this. Instead of needing to exhaust edge cases, OpenJML can prove if our software meets our specifications. This contract-based approach to development can ensure our software is correct and assists others in correctly using our software.

I was motivated to work on this project for two primary reasons. First, this work will

improve aspects of every other computer science discipline as we can create more secure and bug-free software. I enjoyed taking classes such as Systems Software, Discrete Structures, and Programming Languages which all explore topics related to this project. I also was attracted to the more research/academic focus of this project as Dr. Leavens has shared many papers on satisfiability. Second, this work is one part of the open-source movement. Allowing others to contribute and learn from all kinds of codebases allows our discipline to grow quickly. I've learned many aspects of "good" programming by examining open-source repositories.

3.2.3 Broader Impacts

There are use cases for JML in areas like security and embedded systems. An example is Amazon Corretto' Crypto Provider, a performance-oriented replacement to Java cryptographic implementations for Amazon's OpenJDK distribution Corretto. Although Java is not typically the first choice of language for this kind of task, formal verification is relatively easier to implement compared to systems languages like C/C++. With OpenJML, there is potential for Java to be the first step towards widespread industry adoption of formal verification methods. We hope that by contributing to the development of OpenJML we help create tools that make program specification and verification worthwhile.

3.3 Legal, Ethical, and Privacy Issues

This project, in its current implementation, does not introduce any legal, ethical, or privacy implications. The OpenJML project is a free open source (FOSS) project. The common tools for JML are open source as well. The source code for the OpenJML project is licensed under GPLv2 because it derives from OpenJDK. The primary constraint is that all libraries used by the developers to implement this project must be licensable under the FOSS provisions.

Because OpenJML performs analysis on source code, it is important "malware" (such as data-mining) is not introduced in any capacity as many private entities use the tool. This is easily checked as all code is publicly accessible on GitHub.

There exist possible future legal and ethical implications in OpenJML as a Free Open Source tool for static analysis of code. It is possible for contributors to introduce malicious code into the project with the purpose of extracting data from users, namely source code. Precedent for a similar kind of exploit exists. In April of 2021, the University of Minnesota was banned

from contributing to the Linux kernel after several malicious patches were submitted as part of unethical research on these kinds of attacks. Open source projects have become the target of these kinds of attacks due to the nature of their development.

Our team has an ethical and moral obligation to our sponsor Dr. Leavens, to the university, and to the OpenJML maintainers, to not introduce any purposely flawed patches to the project, or to act in any way that will compromise the integrity of the project.

3.3.1 GPLv2 - CE

OpenJML uses OpenJDK for parsing Java files and thus is licensed under the same GPLv2-CE version to prevent any conflicts.

This has two primary requirements:

- The source code must be made available.
- Any changes must be clearly stated.

The 'CE' stands for Classpath Exception. It states that the code is under GPLv2, but anyone who uses the code may use a different license. Revisions to the code is still required to be under GPL, but the code is allowed to be used in a larger system without additional restrictions. This can make it easier for people to utilize the OpenJML tool as part of a larger system.

In general, this license is convenient for this project because all necessary components are open source. All contributions will live under the GPLv2 license and will be pushed to the OpenJML repository at the end of the project. If there is still development work to be done, the code will be pushed to a non-mainline branch. In addition, members are encouraged to continue working on the OpenJML codebase after the completion of this project to help maintain and move JML forward in the software verification community.

4 Project Details

4.1 Goals and Objectives

The primary goal of this project is to lay the groundwork for development and implement five quantifier comprehension expressions namely \max, \min, \num_of, \product, and \sum within OpenJML. A comprehension expression is a function that binds or combines a collection of elements. For example, \sum can combine a collection of array integers and add them together. We seek to achieve this by implementing a series of steps that add the quantifier comprehension implementation to OpenJML.

Satisfiability Modulo Theory is a highly theoretical section of computer science that requires a significant understanding of discrete mathematics and formal logic. Given this, we have a focus on developing this understanding of the theory that allows OpenJML to check program correctness. The goal is not just to implement, but to understand the reasoning for implementation and the core of the problem at hand.

With a working knowledge of the theory, we need to gather an understanding of how SMT is used to verify program correctness. One step up from the core theory is the implementation of SMT solvers like Z3. While we will not be working within Z3, we will be constructing programs within the language that the Z3 solver reads known as SMT-LIB. With this, we will be able to communicate with the solver to verify if statements are satisfiable or not.

It does not matter if we can communicate with the solver if we are unable to give the solver any real cases to compute. This is where conversion from OpenJML to SMT-LIB comes in. We need to develop an understanding of JML as a language and how it compares to a Java program. If we can develop an understanding of how a Java program can be proven correct using JML, we will be more capable of contributing to the conversation about formal verification. This will also give us the means to utilize our knowledge of SMT as it can be used to verify java programs.

With a strong fundamental knowledge of the theory and technologies used in OpenJML, we will contribute to the project. In creating test cases that will be used to implement the five quantifiers using test-driven development we will be building our familiarity with the project and the theory behind it. Our goal is to strive to take this further and implement formal verification of these quantifiers using the tests we have created.

We strive to find understanding rather than produce deliverables. Yes, we plan to produce

tests and code, but this is just a vehicle for us to develop as academics within the field of computer science.

4.2 Requirements

The OpenJML senior design project has three deliverables which can be broken down into sections. Each of these should be completed for each quantifier comprehension that is being implemented. Given that the quantifier comprehensions will need to be implemented differently, the requirements could slightly differ for different quantifier comprehensions.

Member	Quantifier
Colin Herzberg	\sum
Colin Herzberg	\product
Robert Derienzo	\num_of
Robinson Vasquez	\min
Sachin Shah	\max

Table 1: Quantifier Test Work Distribution

Each member will research generic quantifier expression verification and how it relates to the specific quantifier they are responsible for. After the research phase is complete, each member will write all relevant tests for their quantifier. This includes programs that OpenJML should verify and should not verify. Then, each member will hand translate the JML to SMT-LIB format and test with the z3 SMT Solver.

- Develop tests for quantifier comprehensions

  We must first develop test code for JML. We can declare a test to be a java class file containing at least one method which is described with JML annotations. These tests must cover the scope of all five quantifiers. This means we include tests for different use cases of each specific quantifier. At the same time, we need to be sure to cover both cases where the verification will be satisfiable and unsatisfiable.

  - Write java class file containing at least one method described using JML annotations
  - By hand find the correct output which should be produced by JML when running on the test class file

  - Verify that we have covered as many edge cases as possible
  - Create tests for both satisfiable and unsatisfiable cases

- Translate logic into SMT-LIB and test manually on Z3 SMT solver

  To verify that the translations we create from JML into SMT-LIB properly express the logic we want, we must manually test the hand-written SMT-LIB code on the Z3 SMT solver and validate the results.

  - Translate each test case from step one into SMT-LIB code by hand
  - Run the SMT-LIB code through the Z3 SMT solver
  - Verify that the SMT-LIB code produced the expected output.

- Patch OpenJML to generate translations into SMT-LIB from JML annotations (these translations should be comparable to those tested in steps one and two)

  Given the success of steps one and two, we will use the verified tests to develop a patch for OpenJML. Using test-driven development, we will implement all five quantifiers with the static checker.

  - Implement static checking for all five quantifiers
    * Translation from JML to SMT-LIB (intermediary steps of conversion into extended Java with JML and basic block form)
    * Pass SMT-LIB to SMT solver
    * Return useful response generated from the solver's response
  - Implement run-time assertion checking for all five quantifier
    * Create Lambda stubs that will can be used to run the quantifier body
    * Execute the lambda in accordance to the specification
    * Compare results of the lambda versus the java output
    * Return appropriate output to warn user in case of errors

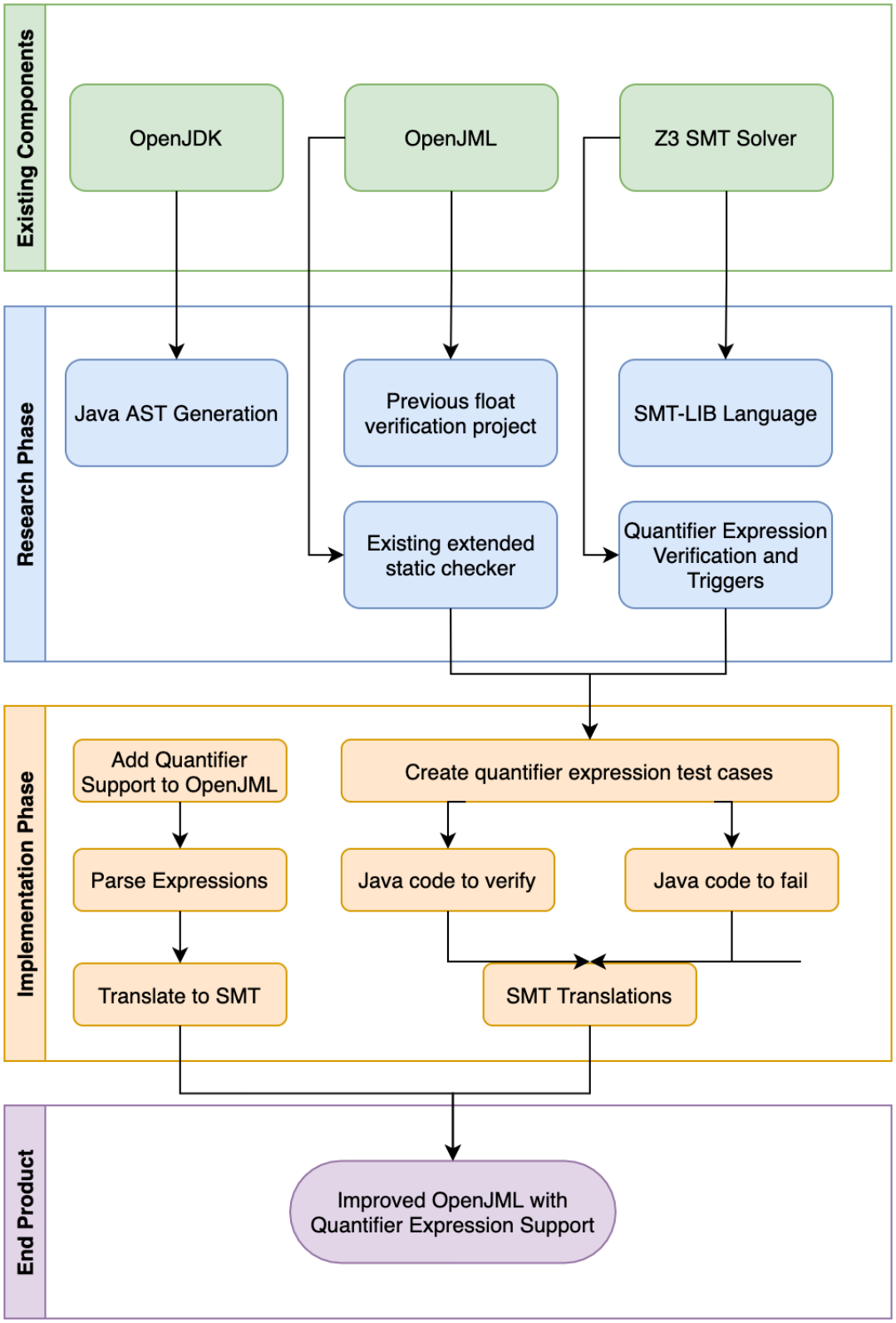


Figure 2: Project Plan

It is worth noting that after scope changes we will be focused on static checking over run-time assertion checking (Figure 2). We will only need to implement these changes for three of the five quantifier comprehensions. The development of the static checking will be focused primarily on creating a logical map from the given quantifier to easily programmable concepts within SMT-LIB. Given the existing infrastructure around translations from JML to SMT-LIB, we will not need to add much, if any translation to the tool.

4.3 What Is a Successful Project?

Through correspondence with Dr. Leavens (sponsor) and Dr. Cok throughout the project, we've defined multiple levels of a successful project. We've defined them as follows:

1. Successful Project

   This is our minimum viable product. If we meet these goals we will be content with the results.

   - Create Java test cases for all five quantifiers

     We want to have a robust testing suite for the implementation of the quantifiers. This will include test cases that test the system in obscure edge case situations. It should also include some example of real-world applications of the quantifiers. Lastly, there must be examples for both satisfiable and unsatisfiable code.

   - Create SMT-LIB test cases that reflect all of the Java plus JML test cases

     This step should provide an example of the SMT-LIB code that the extended static checker needs to produce. At the same time, we will run the SMT-LIB code through the Z3 SMT solver and produce an output. We should find this output to match our expected result on the test case.

   - Verify with Stakeholders that the test cases have coverage over viable use cases

     For this step to be complete we need confirmation from the stakeholders that the tests provide adequate coverage for the implementation of the quantifiers within the code-base. This will be important to ensure the use of test-driven design during the implementation phase.

2. Very Successful Project

   At this stage, we have demonstrated a strong understanding of the problem at hand and implemented a portion of the solution using our tests and research.

- Minimal Successful Project complete
- Implement the quantifiers within OpenJML to be used with the Extended Static Checker.

  Implementing the Extended Static Checking on the three quantifiers (\product, \sum, \num_of) requires extensive knowledge of both OpenJML and SMT. Because of this, completing this implementation will be sufficient for a very successful project.

3. Exceedingly Successful Project

   If we complete this stage the project would be considered Exceedingly successful. We would have built enough understanding to be able to implement the solution, but also will have implemented the entire solution within the project time frame.

   - Very successful project requirements completed
   - Implement bug-fixes and improvements to OpenJML

     Through discussion with Dr. Cok, we have outlined several additional bugs and desired features in OpenJML. If our project proceeds with exceeding success, we will begin working on these problems. They are as follows:

After establishing goals and scope, there was a shift to the project. After meeting with the principal developer of OpenJML, Dr. Cok, we shifted the project to better fit the needs of openJML. There has already been a mostly complete implementation of run-time assertion checking for the five quantifier comprehensions. There is some work to be done to test and fix issues with complex ranges, but the priority of this issue is not as significant as other problems. Within the extended static checker, there has already been a completed implementation of both \max and \min.

With the above information, we have shifted our ideas of success to reflect the new project scope. We will still focus on test cases and translation first and foremost. However, our next goal will be to make progress within static checking of the remaining three quantifiers. The progress within this implementation will determine the level of success achieved.

4.4 Constraints and Limitations

4.4.1 Constraints

Java

- The nature of openJML requires the use of Java, limiting both the scope of the project and the potential workload.
- Java is a rapidly expanding language used by millions of people. This means it develops quickly and as a result supporting tools are hard pressed to keep up with the development of the language. This is especially true of open-source projects like OpenJML. When OpenJML was created, Java was chosen as an alternative to C++ because it was generally simpler and quoting Dr. Leavens "this is no longer the case..." . It is, as we will get into later, of great importance that the portions of code we do contribute are able to be easily maintained in the future.

JML

- To maintain the scope of our project, as well as to complete successfully and in a timely fashion we ought to utilize the portions of JML that have already been implemented. This does limit us from using certain test-cases, quantifiers, etc.
- JML is still in development. This requires us to work in parallel with other teams (most likely without direct communication) to ensure we/they are up to date
- It is required that the work we do is in line with the vision of OpenJML, which is to see more wide-spread adoption of software verification. To accomplish this nothing, we implement should cause confusion or frustration to the end user.

SMT

- Although the nature of our constraint is fairly lax, in that we do get a choice (we as in the end OpenJML user) of which SMT solver to use we are limited by the fact that we must use a compatible one.

OpenJML

- OpenJML is still in development. This requires us to work in parallel with other teams (most likely without direct communication) to ensure we/they are up to date
- As an intermediate between both Java and an underlying SMT solver, openJML is constrained by the general constraints of both the SMT solver and Java

Time

- It is work noting that this project is a requirement of Senior Design which implies there is a hard deadline. Work must be completed by this date for a successful project. There will be no extensions.

Project Related constraints

- The project should be able to be continued without much effort from another senior design team, researcher or professor
- To submit a pull request to openjml the newly added code must be robust, debugged and commented well
- The project must be completed to a satisfactory level to receive a passing grade for Senior Design and eventually, graduate
- Also worth noting that this project is a sponsored one, and as a result we are restricted to developing (at least in the most part) what is desired by our sponsor and his associates.

4.4.2 Limitations

Java

- Direct limitations as a result of using java are few and far in between. Java is a feature rich object-oriented language with millions of users. The only limitation herein is the fact that we MUST utilize Java
- Java utilizes the JVM to run it's compiled bytecode, so are development target is limited to those machines that can configure and run the JVM successfully. This would include most machines as of today, but the possibility for micro-machines to lack this capacity exists.

JML

- We ought to limit ourselves to what JML has currently specified. Additional features may cause disparity between openJML and the specification language JML. The end result of this would be user confusion and unorganized documentation.
- Specifications should be about functional behavior involving data values or other interesting qualities. The cost of this however, is that a human, for the most part, must write the specification. With this in mind, usability is the first priority.

SMT

- Some problems come at the expensive of a great amount of computing resources. This does not necessarily mean they are unsatisfiable, just that it is too costly to satisfy them. [6]
- A number of decidable theories are largely excluded from a majority of SMT solvers because they lack sufficient users to justify the effort in adding support for them. [6]
- SMT solvers are bound by their own set of constraints generally the backend that they use. There are different backends for each different solver, but many use an underlying SAT system which can be quite slow computationally speaking.
- SMT-LIB lacks a way to express the notion of a summation and thus we are limited in this aspect. A custom method for expressing summations must be composed in SMT-LIB for a SMT solver to be able to solve it.

OpenJML

- OpenJML is as we understand it still in development. This requires us to work in parallel with other teams (most likely without direct communication) to ensure we/they are up to date
- As an intermediate between both Java and an underlying SMT solver, openJML is constrained by the general constraints of both the SMT solver and Java
- Although openJML doesn't directly implement the universe type system it is designed in spirit of it. We are thus limiting ourselves to also design our quantifiers in spirit of the universe type system.
- As of yet we cannot deal with concurrency in OpenJML

Time

- Time is a limited and valuable resource, with only a handful of months to deliver this project, it is our most crucial.
- This goes even more so for our advisor, Dr. Leavens and his associate Dr. Cok who have both agreed to assist us going forward; for this we are grateful.

Specification

- Although the specification may prove correct this does not guarantee complete functional correctness of design. The user is perfectly capable of specifying a satisfiable statement that does not stand parallel to his intent. [2]
- In the same vein, the process of verification adds additional complexity to a piece of software, leading to errors in either the verification or the software. This can have a trickle-down effect where an incorrect specification leads to an incorrect implementation
- Specifications can also be arbitrarily broad, proving in more cases than intended. The user would have to be specific enough, so that only intended behavior proves.
- Specifications can be too narrow and focus solely on the code that is written. This means it will prove correct, because it is designed to do so based on the physical code. This however does not mean, that the code itself is fulfilling its intent.

4.5 Project Ideas

4.5.1 Colin Herzberg

In the case of this project, the solutions should not be complex to find. The complexity will be in understanding the problem enough to properly create tests for the requirements and then using this knowledge during the implementation of the solution. The problem at hand (creating tests and using them to implement five quantifier comprehensions) can be split into two phases.

Phase 1: Develop tests (SMT-LIB and OpenJML)

Phase 2: Implement in OpenJML using the tests

Within phase one, there is little to no implementation work to be done. The problem is straightforward and asks to be completed in one way (by hand). The only place where we can get creative is in syntax. We devised a simple, efficient way to write open JML syntax that will represent the quantifiers. Although phase one has simple goals the execution of said goals has proven to be more difficult than anticipated. In support of an effective execution of phase one we extensively researched both SMT-LIB and existing OpenJML annotations. On top of this, our team identified as many edge-cases as we could and this process is always difficult.

My idea on how to solve this will be to split the quantifiers up so that each resource will be developing tests for one specific quantifier. From the past OpenJML teams, we have heard it is best to make sure everyone has domain knowledge in both SMT-LIB and JML. From a division of work perspective, splitting by quantifier sounds inefficient, when we could create subject matter experts in SMT-LIB and JML. However, the main constraint on this project will be understanding rather than development time. By splitting on the quantifiers, we will all be more capable of understanding the pipeline that will need to be created during phase two.

Phase two will be more work but should be accelerated by the domain knowledge we have all gained from phase one. Here I think we would break out of the single quantifier group style and focus on specific subjects. The pipeline of OpenJML static checking starts with user-written Java code with JML annotations. The first step will be converting that to an extended format of Java and JML. This step should take one or two teammates. Following, this is converted to a basic block form. This step will most likely take two teammates. After, we convert to SMT-LIB code and pass it to the solver. This step should take one or two teammates. Finally, we read the response from the solver and give the user useful insights. If successful with static checking, we will begin implementation on the run-time assertion checker.

In this phase, it will be better for us to stay within a domain. Each step on the pipeline is created from a different section within the codebase. By keeping authors in their sections, we can ensure that there is consistency in each step. The benefit of splitting by quantifier on phase one will be extremely noticeable in phase two. As we pass data from one step of the pipeline to the next, we will all have enough understanding of the process to ensure that we can communicate with one another. This project is unlike many other senior design projects where you might have a designer, front-end, back-end, and database team, or even a data pre-processing, and machine learning team. In our case we are not solving a big problem, the scope is small, however, the base knowledge to properly execute is significant.

4.5.2 Sachin Shah

One key aspect of this project is understanding how to formally prove statements with quantifier expressions. Using SMT solving tools such as z3 may assist in this process. Starting with SMT-LIB statements may better highlight the kind of expressions that should and should not verify. In addition, reading the academic work in the formal logic field such as Leino and Monahan in 2009 will provide us a robust foundation for proving quantifier expression statements.

When we develop the OpenJML tests and hand-write SMT translations we should explore automated SMT verification testing outside the OpenJML framework to ensure our translations are valid. This may be specifically important for pieces of SMT-LIB statements that are shared between tests. Another direction to explore is the ability to "optimize" the SMT-LIB statements to reduce the execution time of SMT Solving.

In terms of implementing quantifier support in OpenJML, we should follow the standard compilation pipeline built into OpenJML. Our patch can expand this pipeline in key areas to allow for these kinds of expressions. We can use OpenJML's verbose mode to verify the pipeline is valid along all stages of compiling. Frequently running our patch on existing OpenJML tests and the new quantifier tests will be useful. We should also consult the project's main contributor, Dr. Cok, for coding standards to ensure we follow the best practices. In addition, asking "real-world" users of OpenJML may help highlight pain points we might want to focus on.

4.5.3 Robinson Vasquez

I believe that understanding the theory behind the problem will allow us to much more easily solve the problem. Thus one of the first ideas that comes to mind for a solution for translating the Java with JML specifications to SMT language is going through the Leino and Monahan paper that explains how to translate Specifications from F# to a language a SMT solver can understand. I believe understanding the specifics of this paper will help us greatly in understanding concretely how to implement quantifier comprehension in JML.

We should talk to experts like Dr. Cok for guidance. If we get stuck in some part of the problem or we do not understand some section of JML that we need to understand to advance we should seek help from those that understand both the code and the theory very well.

When it comes to specific solution ideas, I believe that learning how to implement recursion

in the SMT-LIB syntax will be a good step towards finding a good representation for our quantifiers in the SMT-LIB. Once we have an idea of how to represent the quantifiers in SMT-LIB we will have an encoding and we will need to develop a specific encoding algorithm that can be encoded as a program. The reason I believe recursion can help us is that our quantifiers quantify over many numerals and the number of numerals it quantifies over is not fixed, so we will have to define a function that takes a data structure like an array and recursively loops over the data structure applying the quantifier to all its members. We will also need to implement matching triggers to instantiate the quantifiers with terms that we want it to instantiate. I also have the idea of declaring algebraic data types to work as data structures to hold the numerical inputs of our quantifiers.

4.5.4 Robert DeRienzo

The first step in developing a solution is understanding the prerequisites to even beginning the project. This consists of a fundamental understanding of the theory behind the individual quantifiers, the OpenJML codebase, and the SMT language.

The theory behind the quantifiers themselves should be neatly summed up both from a logic perspective and a programming language perspective. Researching OpenJML is a fairly time-consuming task and would consist of dissecting their source code to understand how the different parts interact with one another; this step our team has already begun.

Concerning the SMT language, I've found several decent papers on the matter namely the SMT-LIBv2 Language and Tools... [7]. This paper in particular also has a brief introduction to the type of logic implemented by SMT solvers but more importantly, it provides a good list of research points to become more familiar with the source material.

The next step to fill in our gap of knowledge would be to speak to the authors of the papers we found useful and our advisor Dr. Gary Leavens. By using their breadth of already established knowledge we can accelerate our understanding and development speed. Most of the above is related to research and I feel this is the nature of our project.

I think understanding points A-Z is the most important part of implementing any sort of solution. From my rudimentary understanding, our team needs to specify these quantifiers through JML, using their already established syntax for the 5 missing quantifiers. Then we feed them through a SMT solver, translating them by hand to get a better understanding of what JML does to make them SMT compliant. This is most likely done by some sort of parsing module in JML which we would need to utilize and possibly expand.

Parsing is one thing, yet the actual verification is done by the respective SMT solver so the understanding of this part only needs to be as deep as required by the project.

4.6 Distribution of Work

Our project will build upon the existing OpenJML codebase including OpenJDK and will use the z3 SMT Solver backend (Figure 2). During the research phase, every project member will learn about 3 primary components:

1. The OpenJML Extended Static Checker
2. SMT-LIB Language and Solving
3. Quantifier Expression Verification (based on the work in [5]).

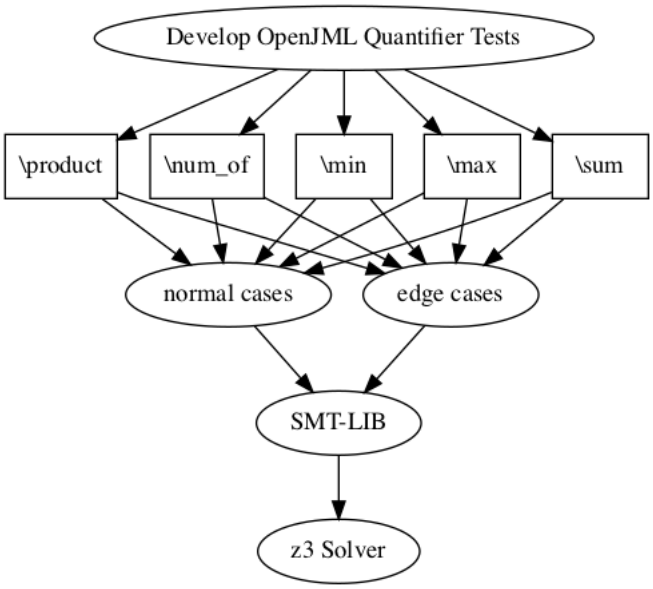


Figure 3: Quantifier Test Breakdown

Although each member should have a working understanding how the OpenJML's extended static checker verifies projects (5.5), we will divide the specific JML to SMT-LIB translation per quantifier (Figure 2).

Each member will research generic quantifier expression verification and how it relates to the specific quantifier they are responsible for (Table 1). After the research phase is complete,

each member will write all relevant tests for their quantifier. This includes programs that OpenJML should verify and should not verify. Then, each member will hand translate the JML to SMT-LIB format and test with the z3 SMT Solver.

Member	Functionality
Colin Herzberg	Expanded Java
Robert Derienzo & Robinson Vasquez	Basic Block Form
Sachin Shah	Handle sat/unsat

Table 2: Extend Static Checker Work Distribution

After each test is created, we will reevaluate the distribution of OpenJML implementation work. Currently, each member will be responsible for implementation the SMT translations for their quantifier in addition to Table 2.

Due to unfortunate circumstances with a prior team member, the distribution of work was shifted. A list of specific responsibilities as related to this document are below:

1. Robert DeRienzo
    - Number of quantifier - Implementation of the \num_of quantifier and relevant design documentation.
    - SMT-Background - Compose SMT Background in the design document
    - Limitations and Constraints - Compose Limitations and Constraints in the design document
    - Division of Labor - Compose Division of Labor in the design document
    - Personal Motivation - Compose Personal Motivation in the design document.
    - Milestones - Determine and design the Milestones to ensure our team is making steady progress on the project.
    - Project Significance - Compose Project Significance in the design document
2. Colin Herzberg
    - Sum Quantifier - Implementation of \sum quantifier and relevant design documentation
    - Executive Summary - Compose Executive Summary in the design document
    - Personal Motivation - Compose Personal Motivationin the design document.

- Technical objectives and goals - Compose Technical objectives and goals in the design
- Product Quantifier - Implementation of \product quantifier and relevant design documentation

3. Robinson Vasquez

- Minimum Quantifier - Implementation of \min quantifier and relevant design documentation
- Personal Motivation - Compose Personal Motivationin the design document.
- Team Motivation - Compile and compose Team Motivationin the design document.
- First Order Logic - Research and compose the relevant documentation for Team Motivation in the design document
- Broader Impact - Compose Broader Impact in the design document

4. Sachin Shah

- Project Manager - Manage efficient workflow for all teammates, manage deadlines and delegate tasks for the project
- Personal Motivation - Personal Motivation - Compose Personal Motivationin the design document.
- Max Quantifier - Implementation of \max quantifier and relevant design documentation
- SMT Solvers - Research and construct relevant design documentation regarding quantifiers in Z3 and alternative SMT Solvers
- Legal, Ethical, and Privacy Issues - Compose Legal, Ethical, and Privacy Issuesin the design document.
- Budget and Finance - Compose Budget and Finance in the design document

4.7 Timeline

4.7.1 Timeline Description

Below we have a simple timeline following the progress of our project. Immediately following this table is a more detailed table, breaking down our progress into more atomic steps.

Table 3 Timeline

Oct. 02	The beginning of this design document. At this point we were expected to have 20 total pages done, and a plan for moving forward. Complete
Oct. 16	Configure Local Development Environment - Running test-cases, and developing on openJML requires that our local environment can use openJML Complete
Oct. 31	Basic development of OpenJML Tests - Once the OpenJML tools have been configured we set out to implement some basic test cases to explore the functionality of OpenJML. Also at this stage we planned some simple test-cases for the quantifiers as well Complete
Dec. 04	Each person is responsible for 30 pages in the design document which concluded the Senior Design 1 Final design document Complete
Jan. 22	Complete Quantifier Test Cases - Based on earlier work, expand a suite of test-cases allowing for full or near full coverage Complete
Feb. 05	Develop SMT Conversions by Hand - Before we can implement the actual code we must first conceptualize how these quantifiers are to be converted to SMT and how a pipeline can be constructed from this Complete
Mar. 06	Begin development of openJML Implementation - Upon the completion of the SMT conversions, and the subsequent pipeline the clear next step is to develop openJML in such a way that it can implement this pipeline. Complete
Apr. 02	After a full code refactoring and review the final goal of the project is the pull request Complete
Apr. 09	Prepare and give Final Presentation Complete

4.8 Facilities and Equipment

To complete this project, we are very lucky to require little in regards to equipment. The only facilities and equipment needed to adequately complete this project are as follows:

- Internet Access - To collaborate with each other and meet with our project sponsor in a manner respectful of current CDC Social Distancing guidelines
- Personal Computer - It goes without saying that a personal computer is required to develop and plan our project.
- Zoom - Our primary method of communication with our sponsor and other stakeholders.
- Discord - Our primary method of communication with one another. Discord is notably effective in that it provides a unified place to organize communication in different threads/channels. It also allows the project manager to directly reach all group members at once.
- Github - The repository for OpenJML is stored on Github and our development progress is as well
- Overleaf - Our document is composed in LaTeX and Overleaf provides an efficient method of writing as a team using LaTeX.
- Google Drive - Google Drive provided a space to collaborate on peripheral documentation related and unrelated to our design document. Including availability, division of labor and so on.
- UCF Library accounts - Our UCF library accounts have given us access to many of the papers which we have used to conduct research.

4.9 Project Budget and Financing

Due to the nature of this project, the team does not require a budget or financing. OpenJML and the SMT solver Z3 are both Free Open Source Software and do not require any specialized hardware to run. The OpenJML Eclipse editor of choice is also free and open-source enabling a consistent work environment for the OpenJML contributors.

Papers related to JML and of use for research by our team are available on Dr. Leavens' JML page. Most are either linked directly or hosted on free publications. The UCF library gives us access to many of the papers that would otherwise not be free, saving us another expense.

We have considered a few expenses and their value. For example, we wrote our design documents in LaTeX. For easy sharing, we used Overleaf. With this tool, we have source control, as well as real-time collaboration. We have considered purchasing the paid subscription for $8.00 per month. With this, we would be able to have more edit history, as well as collaboration features such as comments. We decided against this, using the free tool has worked well, and we have found workarounds for some of the features.

5 Technical Details

5.1 SMT-LIB

5.1.1 SMT LIB Logic

The SMT-LIB format is a standard that defines a formal language, formal rules, commands, and semantics that SMT solvers can implement. The standard is used by SMT solvers like z3 for automatic theorem proving. Satisfiability Modulo Theories (SMT) is a decision problem that asks whether a logical formula is satisfiable given some background theory is true, A formal theory in mathematical logic is usually defined as either a set of sentences or as a collection of models that satisfy certain sentences. So, some SMT solvers are used to check whether assertions made in the SMT-LIB format are satisfiable given certain background theories. The SMT-LIB format has an underlying logic that defines its syntax and semantics which can be used to define background theories in a SMT solver and to write formulas accepted by the SMT solver.

The underlying logic of the SMT-LIB format is many-sorted first-order logic with equality. This is a formal system whose syntax consists of logical and non-logical symbols. The logical symbols are symbols that are part of every many-sorted first-order language and these consist of

- $\wedge, \vee, \implies, \iff, \neg, \forall, \exists, =$ and grouping symbols like () and,
- Variable symbols like x,y, etc

The non-logical symbols consist of Predicate symbols, Function symbols, constant symbols, and sort symbols. Many sorted logics also consists of a grammar, which are rules that define which formulas are well-formed, and inference rules that dictate what formulas are derivable given other formulas have already been obtained as previously stated in the formal logic section of this document.

As previously stated the semantics of Many sorted logic can be defined using Model theory. It can be used to define how the logical connectives affect the truth value of the formulas it is operating over and to define the meaning of the predicate symbols, of the function symbols, of the constant symbols, and the sort symbols. Sorts can be seen as types in programming.

A many-sorted first-order theory is a set of well-formed formulas

We say that a well-formed formula ϕ is valid given some background theory T if and only if ϕ is true in every model that makes all the sentences in T true. We say that a well-formed formula ϕ is inconsistent concerning some background theory T if and only if ϕ is false in every model that makes all the sentences in T true. We say that a well-formed formula ϕ is satisfiable for some background theory T if and only if ϕ is true in some model that makes all the sentences in T true.

What a SMT solver does to see whether a formula ϕ is valid is by seeing if the negation is satisfiable. If it is satisfiable then ϕ is not valid, if it is unsatisfiable that means that $\neg\phi$ is inconsistent with T and this means ϕ is valid. (this only works for logics that are closed under logical negation)

Now it is worth noting that SMT-LIB changes some things from the ordinary many-sorted first-order logic. First, all predicate symbols and constant symbols in ordinary many-sorted first-order logic are just function symbols in the SMT-LIB standard. Constant symbols in SMT-LIB are just functions that return a constant and take no inputs. Predicate symbols in SMT-LIB are just functions that return a Boolean sort (Boolean type) this implies that in the SMT-LIB standard there is no syntactic distinction between terms and formulas.

In ordinary many sorted first order logic, terms are defined recursively in the following way:

- If t is a constant symbol, then t is a term
- If t is a variable then t is a term
- If t is $f(t_1, t_2, \ldots, t_n)$ where t_1,t_2,...,t_n are all terms and f is a function symbol then t is a term

Formulas are rather defined recursively in the following way:

- If ϕ has the form $t_1 = t_2$ and t_1 and t_2 are terms then ϕ is a formula
- If ϕ has the form $R(t_1, t_2, ..., t_n)$ where R is a predicate symbol and $(t_1, t_2, ..., t_n)$ are all terms then ϕ is a formula
- If ϕ has the form $\alpha \implies \beta$ or $\alpha \wedge \beta$ or $\alpha \vee \beta$ or $\neg\alpha$ and α and β are formulas then ϕ is a formula
- If ϕ has the form $\forall x_\sigma \alpha$ or $\exists x_\sigma \alpha$ and x is a variable of sort σ and α is a formula then ϕ is a formula

In contrast to ordinary many-sorted first order logic, in SMT-LIB formulas are terms of the form $f(t_1, t_2, ..., t_n)$ where the arity of f is $\sigma_1 \times \sigma_2 \times, \ldots, \times \sigma_n \rightarrow Bool$ where $\sigma_1 \times \sigma_2 \times, \ldots, \times \sigma_n$ are the sorts of $t_1, t_2, ..., t_n$. So formulas are just terms of a boolean sort.

In SMT-LIB there are also sort terms in addition to sort constants. In ordinary many-sorted first-order logic, there are only sort constants. Although sort terms in SMT-LIB are reducible to combinations of sort constants. So those are two of the differences between the SMT-LIB standard and ordinary many-sorted first-order logic.

The SMT-LIB standard allows specifying background theories for the solvers. Background theories help interpret certain function symbols the solver might want to have predefined, they effectively act as the models the solvers consider when evaluating commands. Models in SMT-LIB are somewhat more complicated than models in classical many-sorted first-order logic and will be explained in more detail in the logical semantics section of SMT-LIB. SMT-LIB differentiates between two types of background theories, basic theories, and combined theories. Basic theories are theories explicitly defined in the SMT-LIB catalog which include for example the theory of real numbers. Combined theories are theories formed from basic theories via a combination operator.

The SMT-LIB interface defines the textual interface for SMT solvers which allows for input and output to the screen, checking satisfiability, showing proofs, showing models all of which are command-based.

5.1.2 Logical Semantics of SMT-LIB

To better understand the syntax and semantics of the SMT-LIB language, it is convenient to talk about them in a more abstract mathematical manner first. That is where the logical semantics of the language comes in, which shows one way in which the SMT-LIB format can be formally represented. The logical semantics defines the theoretical background of the SMT-LIB format. The abstract syntax of the SMT-LIB consists of various infinite sets:

- an infinite set S of sort symbols s containing the symbol Bool. Sorts are like types in programming languages, an example of a sort symbol in SMT-LIB is Int
- an infinite set U of sort parameters u which are sorts or arbitrary sorts that can serve as parameters to parametric sorts
- an infinite set X of variables x which would represent variables, meaning arbitrary placeholders for terms

- an infinite set F of function symbols f containing the symbols $=$, $\wedge$, $\neg$ functions take 0 or more parameters and return a term
- an infinite set A of attribute names a, attributes are descriptions that can be attached to terms that can help solvers interpret or which just functionally work as comments in programming languages
- an infinite set V of attribute values v which represent values that can be given to some attributes.
- the set W of Unicode character strings w
- a two element set $B = \{true, false\}$ of Boolean values b, this set is necessary for evaluating satisfiability in SMT-LIB, satisfiability is a concept that depends on the concept of truth
- the set $\mathbb{N}$ of natural numbers n
- an infinite set T of theory names t to identify different theories
- an infinite set L of logic names l to identify different logics

Now we will define and explain various concepts in the formalized SMT-LIB Format

Definition of sorts

For all non-empty sets $Su \in P(S)$ meaning the subsets of S, and all mappings are: $S \to N$, The set $Sort(Su)$ of all sorts over Su is defined as follows:

- every $s \in Su$ with $ar(s) = 0$ is a sort, meaning every sort symbol defines a sort
- if $s \in Su$ and $ar(s) = n$ and $n > 0$ and $\sigma_1, \ldots, \sigma_n$ are sorts then the term $s\sigma_1, \ldots, \sigma_n$ is also a sort, meaning a sequence of sort symbols is also a sort, representing a parametric sort

ar represents the arity of the sort which means the amount of sort parameters it takes.

An example of a parametric sort with multiple parameters that can be used in SMT-LIB is Array Int Int. This sort represents an array that has an index of sort int and whose members are of sort int, another variation would be Array Int Bool, which would be an array of an index of sort int and whose members are of sort Boolean. .

Definitions of terms

Terms are built from X, F, and a set of binders.
Binders are symbols that bind variables to a particular behavior. Binders bind variables to terms. The following binders are the binders defined in SMT-LIB:

- $\exists x_1 : \sigma_1, \ldots, x_n : \sigma_n$ is a sorted existential binder for every variable in the scope of the binder. The binder is joined to a formula and forms a Boolean. This binder binds variables to the boolean term that follows it and can be seen as a binding between the variable and the constant symbols
- $\forall x_1 : \sigma_1, \ldots, x_n : \sigma_n$ is a sorted universal binder for every variable in the scope of the binder. The binder is joined to a formula and forms a Boolean
- Let $x_1 = t_1 \ldots x_n = t_2$ *in* t is a let binder for all the variables in the scope. This binder replaces each free occurrence of $x_1, \ldots, x_n$ by $t_1 \ldots, t_n$ in the term t
- $Match_with\ p_1 \rightarrow _ \ldots p_n \rightarrow _$is a match binder for the variables that occur in the patter p_i

In SMT-LIB the binders are the only logical symbols of the Formal system, usual logical symbols like the material implication symbol ($\rightarrow$), the and symbol ($\wedge$), the negation symbol ($\neg$), and the or ($\vee$) are all functions whose output is a Boolean and are defined by an instance of a background theory called core. The core theory is one of the most essential theories for SMT solvers to implement and will be explored further in other sections of this document.

Annotations: Attributes are annotations, meaning information for the reader to understand the meaning of the term that is being annotated, to the solver a term being annotated or not with attributes makes no difference so they are like comments.

Definition of signature

A SMT-LIB signature is a tuple Σ which consists of :

- a set $\Sigma^S \subseteq S$ of sort symbols containing Bool
- a set $\Sigma^F \subseteq F$ of function symbols
- a finite set $\Sigma^C \subseteq \Sigma^F$ of constructor symbols these are functions that construct instances of algebraic datatypes

- a finite set $\Sigma^G \subseteq \Sigma^F$ of selector symbols, disjoint with Σ^C that are functions which can output constructor parameters
- a finite set $\Sigma^T \subseteq \Sigma^F$ of tester symbols which are functions that test whether a certain function of a certain algebraic datatype was created by a specific constructor
- a total mapping $con_(\Sigma) : \Sigma^S \to P(\Sigma^C)$, which assigns a subset of the set of constructors to sort symbols, these are the constructors that can construct instances of this sort, meaning the functions that create terms of that particular sort. These sorts would just be algebraic datatypes
- a total mapping $ar : \Sigma^S \to \mathbb{N}$ which assigns an arity to each sort and represents the number of sort parameters this sort can take
- a total mapping $sel_(\Sigma) : \Sigma^C \to (\Sigma^G)^*$ where $(\Sigma^G)^*$ is the set of all possible sets of selectors represented as a sequence. This assigns n selectors to each constructor of arity n so that each selector is mapped from one and only one constructor
- a bijective mapping $tes_(\Sigma) : \Sigma^C \to \Sigma^T$ which assigns a tester to each constructor, this is the tester that tests whether a term of sort $s\sigma$ was created by their corresponding constructor
- a partial mapping $:X \to Sort(\Sigma^S)$ which assigns a sort to some variables in X, since we want to bind some variables to sorts to represent an arbitrary member of that sort. This is helpful for binders like the existential quantifier or the universal quantifier
- a ranking relation $R : \Sigma^F \times Sort(\Sigma^S)^+$, so that the tuple $(f, \sigma_1, \ldots, \sigma_n\sigma)$ means that f takes as inputs terms of sort $(\sigma_1, \ldots, \sigma_n)$ and outputs a term of sort σ

The ranking relation satisfies all of the following assertions:

- for all constructors $c \in \Sigma^C$, selectors $g_1 \ldots g_n = sel_\Sigma(c)$ and sorts $\sigma_1, \ldots, \sigma_n, \sigma \in Sort(\Sigma^S)$ if $(c, \sigma_1, \ldots \sigma_n\sigma) \in R$ then $(g_i, \sigma\sigma_i) \in R$ for all i from 1 to n. This means that for all constructors in the constructor symbols set, if the selectors of the constructor are $g_1 \ldots g_n$ and the constructor takes as input terms of sorts $\sigma_1, \ldots \sigma_n$ and outputs a term of sort σ then each selector of c takes as input a term of sort σ and outputs the sort of some of the inputs. For a more concrete example see the syntax section of this document.

- For all constructor $c \in \Sigma^C$ and testers $p = tes_\Sigma(c)$, if $(c, \sigma_1, \ldots, \sigma_n\sigma) \in R$ then $(p, \sigma Bool) \in R$. This means that for each constructor c in the constructor symbols set, if c takes as input terms of sort $\sigma_1, \ldots, \sigma_n$ and outputs terms of sort σ then, the tester t assigned to that constructor, takes as input a term of sort σ and outputs a term of sort Bool. This just means the tester test whether a term is an instance of a constructor for terms of sort σ. If c is a constructor for terms of sort σ then p will return true on terms of that sort. These sorts that are constructed by constructors are algebraic datatypes and will be defined more extensively shortly. If c is not a constructor for terms of sort σ then p on inputs that are instances of c returns FALSE.
- There is no constructor $c \in \Sigma^C$ such that $(c, \tilde{\sigma}_1\sigma) \in R$ and $(c, \tilde{\sigma}_2\sigma) \in R$ for distinct $\tilde{\sigma}_1$ and $\tilde{\sigma}_2$. This means that there is no constructor that outputs instances of a datatype on inputs of different sort. This means constructors have a fixed rank such that to construct terms of a certain sort they only accept inputs of a specific sort. $\tilde{\sigma}$ means a sequence of possibly many sorts

Signatures defined the symbols a model is going to consider. the function and sort symbols that are defined in a model is based on the signature, so if models were viewed as a formal language (which they are not) then the signature would more or less act as the alphabet for that language.

Sorts in the signature that are assigned one or more constructors, meaning $con_\Sigma(s) \neq \emptyset$, are called algebraic data types, meaning sorts constructed from other sorts. This simulates the concept of constructors and classes in Object Oriented Programming. Algebraic datatypes act as classes and constructors in SMT-LIB act as constructors for classes in object oriented programming. You have algebraic datatypes defined using primitive sorts and have constructors that construct instances or terms of that datatype. It follows that sorts with no constructors act as primitive types.

Sub signature: Let Ω be a signature and Σ be another signature.Σ is said to be a subsignature of Ω if:

- $\Sigma^S \subseteq \Omega^S$, so Ω constains all the sort symbols Σ contains and possibly more
- $\Sigma^F \subseteq \Omega^F$ so Ω constains all the function symbols Σ contains and possibly more
- If $s \in \Sigma^S$ and $ar_\Sigma(s) = n$ then $ar_\Omega(s) = n$, meaning s has the same arity in Σ and Ω
- If $s \in \Sigma^S$ and $con_\Sigma(s) = c_1, \ldots, c_n$ then $con_\Omega(s) = c_1, \ldots, c_n$, meaning s has the same constructors in Σ and Ω

- If $c \in \Sigma^C$ and $sel_\Sigma(c) = g_1 \dots g_m$ then $sel_\Omega(c) = g_1 \dots g_m$ meaning c has the same selectors in Σ and Ω
- If $c \in \Sigma^C$c and $tes_\Sigma(c) = p$ then $tes_\Omega(c) = p$ meaning c has the same testers in Σ and Ω
- For all x in X and $\sigma \in Sort(\Sigma)$ $x : \sigma \in \Sigma \iff x : \sigma \in \Omega$. Meaning that if x is of sort σ in Σ then x has sort σ in Ω, so the variables are assigned the same sort in both signatures
- For all $f \in \Sigma^F$ and $\bar{\sigma} \in Sort(\Sigma)^+$, if $f : \bar{\sigma} \in \Sigma$ then $f : \bar{\sigma} \in \Omega$. Meaning that if f has a rank of $\bar{\sigma}$ in Σ then f has rank of $\bar{\sigma}$ in Ω

Ω is said to be an expansion of Σ

Overloading

Functions in SMT-LIB can be overloaded like functions in programming can be overloaded. Two or more functions with the same function symbol can be given different ranks and according to the sorts of the inputs the function is applied as the function with the respective rank. For example, let's say we have a signature with sort symbols Int, Real, and Array, the sorts of the signature would be Int, Real, Array Int Int, and Array Int Real with their usual meaning. We could have two functions with the function symbol sum. The rank of one of the sum function would be (Array Int Int) Int meaning the input is an array of integers and the output is an integer, let's say that the meaning of sum is such that it returns the sum of the elements of an int array. The rank of the other sum function is (Array Int Real) Real, meaning the input is an array of reals and the output is a real, we could take this function to be the function that takes an array of reals and returns the sum of the array of reals. This would be the application of the concept of overloading in SMT-LIB and shows that this can be done

Well-sorted terms

Well-sorted terms are any terms that are derivable from the rules of the grammar. The terms derivable from the rules of the grammar also depend in the signature which fixes the function symbols, sort symbols, constructor symbols, the ranks of the different functions, which constructors are assigned to which sorts and so on. The terms derivable from the rules of grammar also depend on background theories since the background theory also defines a certain signature. For example, instances of the Core theory and the ArrayEx theory are built into solvers like Z3, this allows one to construct well-sorted terms like :

select (A,i):int which is the abstract representation of the select function in the ArrayEx theory which takes an array A and and an int i and returns the int value of the term stored at position i. There is also $\implies$ (A,B):Bool which is the abstract representation of the material implication function in the Boolean logic instance of the Core theory which takes two terms A and B of sort Bool and returns a term of sort Bool (True or False).

The concrete rules of the grammar indexed by some signature can be found in the SMTLIB standard.

In SMT-LIB scripts are the terms being evaluated by the solver implementing the SMT-LIB standard. As it has been stated before formulas are terms of sort bool and closed formulas are formulas with no free variables. In many logics closed formulas are called sentences. For a formula with variables to be closed it needs to have a binder that binds the variables and evaluates to bool like the existential quantifier binder ($\exists$) and the universal quantifier binder ($\forall$). the assertions evaluated by solvers are closed well formed terms of sort bool.

Definition of structure or model:

A Σ-model (or Σ-structure) A is a 2-tuple (U_A, I) where U_A is a set called the universe of A and I is an interpretation function that interprets in the following way:

- each sort σ is interpreted as a sort $\sigma^A \subseteq U_A$, we call it the domain of σ in U_A, the elements e in σ^A are functions of arity 0 (constants) that are of sort σ^A. So for example let's say we have a signature with the sort symbols Int and Real, without a structure this would be an uninterpreted sort with no fixed meaning so they could represent any sort. Now we give it a model (R,I) where R is the set of Real Numbers. I maps Int to the set Int^A=Z which is the set of Integers and which satisfies $Z \in R$, the members of the interpretation of Z are integers. I maps Real to the whole set R, meaning the interpretation of the sort symbol Real would be the set of real numbers.
- each function symbol of rank 0 (constant) $f : \sigma$ is mapped to $(f : \sigma)^A \in U_A$ and $(f : \sigma)^A \in \sigma^A$. This means that each constant symbol which is just a function symbol of rank of one sort is a member of the interpretation of the sort σ. So for example if the function symbol is 5, I maps 5 to the integer 5 or the Real 5 which is a member of the Reals or Integers, depending on the mapping.
- each $f : \sigma_1, \ldots, \sigma_n\sigma$ is mapped to a total function $(f : \sigma_1, \ldots, \sigma_n\sigma)^A$ which is a function of the form $(f : \sigma_1 \times \ldots \times \sigma_n \to \sigma)^A$. The rank of this function would be $\sigma_1 \ldots \sigma_n\sigma$. This just means that each function symbol that has 1 or more arguments and is thus not a constant becomes a function with arity = n over the interpretations of the sort

symbols. So for example if there is an uninterpreted function symbol + with rank (Int Int Int) in the signature it could be mapped by I to a function $+ : Z \times Z \to Z$ in the model which would represent the function that adds two integers and returns an integer and which would have arity of 2.

Reducts:

For a model B with a signature Ω where Σ is a sub-signature of Ω then the Σ-reduct of B is the structure with the same universe as B with signature Σ and that interprets the sort and function symbols in the same way as B, so it is a model that mimics B but that has a smaller signature with possibly fewer symbols

Absolutely free structure:

An Absolutely free $\Sigma-$model A with generators G is a model where G is a subset of its Universe U_A and, there exist some signature which we call Σ_G that is an expansion of the signature Σ by adding every constant symbol c such that its interpretation is in G. A interprets every Σ_G ground term as itself. A ground term is a term with no variables.

SMT-LIB Σ-model is a Σ-model such that

- $Bool^A = \{true, false\}$ Since we want to show the satisfiability of assertions we need the Bool sort symbol to represent the boolean values true and false such that we can use it to define satisfaction in a model.
- let Ω be the signature obtained from removing all the non-constructor function symbols from Σ then the Ω-reduct of A is an absolutely free structure with generators of all the members of non datatype sorts, meaning G is the set of all terms whose sorts are primitive and not created via constructors.
- for all constructors c_i or rank $\sigma_1, \ldots, \sigma_n\sigma$ in Σ and selectors $g_1 = \sigma\sigma_1, g_2 = \sigma\sigma_2, \ldots g_n = \sigma\sigma_n$ with $sel_\Sigma(c_i) = g_1, g_2, \ldots, g_n$ then $g_i^A(c_i^A(a_1^A, \ldots, a_n^A)) = a_i^A$ for constant symbols (functions of no arguments) a. meaning selectors select for the corresponding argument in the constructor. So for example if your data type is a list with a constructor that adds a head to your current list, then selectors allow you to get the head and the previous list.
- for all constructors c_i or rank $\sigma_1, \ldots, \sigma_n\sigma$ in Σ, testers q of rank $\sigma Bool$ such that $tes_\Sigma(c_i) = q$ if f is a function of sort σ^A in the model then $q^A(f^A) = true \iff f \in c_i^A$ meaning the interpretation of testes is a function that when given a term of a certain

sort it tests whether that is an instance created by their corresponding constructor and if it is it returns true, if it is not it returns false

The meaning of terms:

A valuation is a partial function $v : X \times Sort(\Sigma) \to A$ such that :

$$\forall x \forall \sigma : (x \in X \wedge \sigma \in Sort(\Sigma)) \implies v(x : \sigma) \in \sigma^A)$$

.We write the valuation that maps each $x_i : \sigma_i$ to $a_i \in (\sigma_i)^A$ If v is a valuation into Σ-model A, the pair $I = (A, v)$ is called a Σ-interpretation. v gives an interpretation to the formulas with free variables so that for example let the Signature have a function symbol $=$ and a function symbol $+$ and the structure A interprets them in the usual manner, and the universe of A is the set of natural numbers. If the set X of variable symbols contains variable symbols x_1 *and* x_2 *and* x_3 then we can construct the formula $x_1 + x_2 = x_3$. Now since this is a formula with free variables, a model does not give it an explicit interpretation such that the formula can evaluate to true or false, for that we need a valuation v. v can interpret $x_1 \to 2$, $x_2 \to 4$ and $x_3 \to 6$. This valuation would evaluate the formula to be true, thus the formula is satisfiable since there exists a valuation that satisfies it. Now, since this is not a sentence, it is not said to be true under the model A since only closed formulas are said to be true under a model, thus, structures are enough to talk about the satisfiability of a sentence. That a sentence ϕ is true in a model A can be represented as:
$A \models \phi$
This can also be expressed as A semantically entails ϕ. A sentence is said to be satisfiable with respect to a signature Σ if there exists a Σ-model where that sentence is true. If a sentence is false in every Σ-model it is said to be unsatisfiable or contradictory, for example let c be a function with rank Int Int Bool, meaning it takes two integers and returns a Boolean, let a and b be constant symbols then the sentence $(c(a, b) \wedge \neg c(a, b))$ is unsatisfiable, it is a contradiction, because no matter what interpretation is given to c, or a or b, the sentence always evaluates to false. The sentence as expressed in the concrete SMT-LIB syntax would be (assert (and (c a b) (not (c a b)))). Now this is given that we have the Boolean logic instance of the background theory Core which is needed to give a fixed interpretation to the function symbols $\wedge$ and $\neg$ which in specific SMT-LIB syntax would be *not* and *and*. Since the only logical symbols that evaluate to true are the existential quantifier binder and the universal quantifier binder, it seems we need at least one theory to give interpretation to predicate and Boolean symbols in order to construct contradictions in the SMT-LIB language. In classical-many sorted first order logic the symbols $\wedge, \to, \vee and \neg$ are all logical symbols and have fixed meanings across all models. In SMT-LIB we need at least an instance of a theory to fix the meaning of all those symbols.

Isomorphism:

Two structures A and B are said to be isomorphic if and only if there exists and isomorphism between them. An Isomorphism between A and B is defined as a function $h : U_A \to U_B$ such that:

- for all sort symbols σ with interpretation of σ^A in A and σ^B in B, $a \in \sigma^A$ if and only if $h(a) \in \sigma^B$ and there does not exist another $a_2 \in \sigma^A$ that is not equal to a such that $h(a_2) = h(a)$, meaning h maps each member of σ^A to a different memeber in σ^B and the domain of h is all of U_A. This shows that h is a bijection from U_A *to* U_B
- for all functions f of rank $\sigma_1, \ldots, \sigma_n$
  $h(f^A(a_1, \ldots, a_n)) = f^B(h(a_1), \ldots, h(a_n))$
  and again h is a bijection from the outputs of f^A to the corresponding output of f^B.

Isomorphism show that there is a direct map between two models in both terms and behavior, which implies that both structures behave in the same way. This means that one structure can be used to model or predict the other. One example is the model, which we will call T, whose universe consists of true and false and a finite set of sticks all of a different length, whose sorts symbols are Bool and stick, where Bool represents boolean and stick a sort stick to which all sticks are part of. The function symbols consist of f and g both of rank *stickstickBool* meaning they take two sticks and return a boolean. In T f(a,b) returns true if and only if b is longer than a, g(a,b) returns true if and only if a is the same length as b. There is an isomorphism between this structure and a structure Q whose universe is a finite subset of the rational numbers and true and false and whose functions are ¡ and = and have the usual interpretation. the isomorphism h maps each stick to a rational number such that for any sticks a and b if b is longer than a then it is given a bigger rational number than it is given to a. The universe of Q is the same size as the universe of T, true and false are mapped to true and false and h(f(a,b)) = ¡(h(a),h(b)) since if b is longer than a then h(b) is greater than h(a) and if a is longer than b then h(a) is longer than h(b). For the function symbol g(a,b) = =(h(a),h(b)) is always true also since g(a,b) is true if a and b are the same stick and if they are the same stick h(a) and h(b) are the same number and thus equal. This shows that they have the same behavior and Q can be used to model lenght in T using a finite subset of the rational numbers. A homomorphism is the same as an Isomorphism except it no need be a bijection.

Theories:

In SMT-LIB, theories can be seen as a collection or class of models over the signature of

the language that give the same interpretation to the terms as that of the theory and that possibly make the axioms of the theory true if this theory has axioms. For example the theory core is a theory with no axioms and just defines the ranks of certain functions and defines a sort of arity 0 called Bool, it also defines the functions of arity 0 true and false which are of sort Bool. The class of models of the theory core, are all those models that interpret in the following way

- the sort symbol Bool as the set $\{True, False\}$,
- that interpret the functions true and false as functions with arity 0 which are members of $\{True, False\}$
- the functions $\implies$ as a function $\implies : \{True, False\} \times \{True, False\} \to \{True, False\}$
- *and* as a function $and : \{True, False\} \times \{True, False\} \to \{True, False\}$
- *or* as a function $or : \{True, False\} \times \{True, False\} \to \{True, False\}$,
- *not* as function $not : \{True, False\} \to \{True, False\}$
- $=$ as a function $=: A \times A \to \{True, False\}$,
- *Ite* as a function $ite : \{True, False\} \times A \times A \to A$

Where A are some arbitrary sets and it represents an if-then function in the instance of the theory implemented by certain SMT solvers.

Core does not have axioms, but if the theory has axioms then the class of models of the theories are the class of models that give the same interpretation as the theory which would include the interpretation that the axioms are true. So it would be all the models that give the same interpretation to the various symbols and functions of the signature and that would also make the conjunction of all the axioms true.

The abstact concept of solvers like z3 trying to show whether a set of sentences in SMT-LIB are satisfiable can be expressed in the following way: A set of sentences are satisfiable given some background theory T if and only if there exists some model for the theory T that makes all the axioms of the theory true and also makes all the sentences in the set true. If there exists no such model the set of sentences are not satisfiable and given that z3 accepts the law of excluded middle ($(p \vee \neg p)$ is true in every model) the negation of a non-satisfiable set of sentences would be true in every model of the theory. If a set of sentences are true in

every model of the theory we say that T semantically entails the set of sentences ϕ, written $\models_T \phi$

SMT-LIB consists of combined theories and basic theories. Basic theories are instances of theory schemas as defined in theory declarations in the syntax section of SMT-LIB. A theory schema is a way to represent a set of theories. So theory schema declaration defines many theories just not one that are consistent with the schema. For example the theory that interprets the functions and sorts of the core theory the same way propositional logic does is an example of a basic theory which is an instance of the core theory schema. To understand how combined theories are formed from basic theories we first need to understand what compatible signatures are.

Two signatures Σ and Ω are said to be compatible if they :

- Have exactly the same sort symbols
- Have exactly the same constructors, selectors and tester symbols
- have the same arity for their sort symbols and assign the same sorts to their variable symbols

For two theories to be combined they need to have compatible signatures. The combination of two theories T_1 and T_2 is a theory T whose signature consist of Σ_T which is the smallest signature which is compatible with $\Sigma_T 1$ and $\Sigma_T 2$ and which is also an expansion of both $\Sigma_T 1$ and $\Sigma_T 2$ and consists of all the Σ_T-models whose $\Sigma_T i$-reduct is isomorphic to the models of T_i, meaning the models of T when reduced to a sub-signature $\Sigma_T i$ form models that have the same behavior as the models of T_i which are one of the theories in the combination and which have signature $\Sigma_T i$. Combinations of theories are essential for the creation of more complex and powerful theories in terms of expressive power and to create logics.

Logics:

A logic consist of the following:

- a specific signature Σ and a Σ-theory which consists of the combination of various theories
- a set of structures that are the set of models for the Σ-theory meaning the models that interpret the same way as the theory does.

- the set of all sentences is a subset of the set of all Σ-sentences

So a logic consists of a theory with a given signature and all the models of the theories. In this way they seem very similar to theories. As explained in the syntax of SMT-LIB many logics can be formed through theories or the combination of theories.

Now we will talk about SMT-LIB more concretely, we will go over the syntax of SMT-LIB and the semantics of how solvers are supposed to evaluate SMT-LIB scripts.

5.1.3 Syntax and syntactic categories

The syntactic rules of SMT-LIB are expressed as a context-free grammar, which are rules to generate strings that form a context-free language. In other words, SMT-LIB without type-checking is a context-free language. If we include type checking SMT-LIB is a context-sensitive language since it would also have to make sure that well-formed strings follow the typing rules. For example, $5 + True$ could be generated by some grammars, but would not follow ordinary typing rules for the language since summation is defined as a function of numeric types like integers or real numbers.

A context-free grammar consists of variables, terminal symbols, and rules that tell you to what terminal symbols and variables, a variable can be rewritten to. Context-sensitive grammars are very similar, the only difference is that you can have variables in conjunction with other variables and terminals be rewritten into a larger string, so variables have "context" around them when they are rewritten.

SMT-LIB syntax allows you to declare functions, constants, and sorts, where sorts are the type of a variable. This allows one to create uninterpreted functions and sorts which can have different interpretations under different models.

SMT-LIB also allows you to explicitly define functions and sorts and to create new sorts. You can define functions using other functions in the background theory and this is very useful towards expressing the OpenJML quantifiers in interest. The OpenJML quantifiers can be seen as functions in SMT-LIB that have sort parameters and output a term of a numeric sort. For example $(\backslash \text{sum} t_1 i, \ldots, t_n j; B(i, \ldots, j); A(i, \ldots, j))$ can be represented mathematically as a function

$f : R^n \to R$ that takes numbers in the range of the function $A : S^n -> R$ where the inputs to A are n terms of sort $t_1, \ldots, t_n$ respectively and which when inputted into the Boolean

formula B output true.

So SMT-LIB can work as a representation for these quantifiers, the main problem comes from the fact that the number of inputs to these quantifiers is not fixed and is variable.

An example of a function definition in SMT-LIB is the following:

```
(define-fun f ((x Real) (y Real)) Real
    (* x y)
)
```

The above function takes two Real numbers and then returns a Real number which is the multiplication of the two arguments. This definition is possible given that the multiplication function is already defined from a background theory instantiated by the solver.

Theory declarations

Theory declarations allow one to syntactically restrict the models under consideration. By syntactically declaring theories a fixed meaning is given to various terms. The models under consideration for the assert statements would have to be the models that assign the same meaning as the theory declarations. As previously stated, in SMT-LIB you can declare various theories and combine them to define the meaning of various commands. Two very important theories for example that can be declared in SMT-LIB are the Core theory and the theory of arrays ArrayEX.

The Core theory is a theory that defines a theory schema, meaning it defines a collection of theories all of which are consistent with the Core theory. The Core theory defines the Boolean sort as a sort of arity 0 and the 0-arity functions True and False as functions of sort Bool. It also defines the rank of the various functions, mainly the implication function, the and function, the negation function, and so on. In regular classical logic, these are not functions but rather logical symbols that have a fixed meaning overall logics. In SMT-LIB these are functions that output a term of sort Bool and have no specific interpretation, meaning as long as the rank of these functions is respected they could be defined in many different ways. So for some possible models of SMT-LIB, the implication symbol $\implies$ could be defined with the following truth table.

p	q	$p \implies q$
False	False	True
False	True	False
True	False	True
True	True	False

when in regular propositional logic it has the fixed definition of

p	q	$p \implies q$
False	False	True
False	True	True
True	False	False
True	True	True

So these usually logical symbols do not have a fixed meaning even with core as a background theory, both definitions above are consistent with the rank given to the function $\implies$ of Bool Bool Bool in the theory core. Some Solvers like z3 implement the model of core that interprets those functions as regular Boolean logic or classical propositional logic does.

Every theory has a set of theory attributes that are part of a predefined category. There are 7 predefined categories:

- :defintion
- :funs
- :funs-description
- :notes
- :sorts
- :sorts-description
- :values

The most important categories for the solvers are the :funs and :sorts categories. In these, you can declare the functions and sorts of the theory. For sorts, you would give it a name,

which would be the identifier of the sort, and a arity, which would be the arity of the sort. The general form for declaring theory sorts is

:sorts $((s_1 n_1), \ldots, (s_m, n_m))$,

where s_i is the identifier of the sort and n_i is the arity of the sort. For example for one theory one could declare

:sorts $((Matrix \quad 3))$

which would be a sort for a 2 dimensional matrix for a theory of matrices over $R^{m \times n}$, meaning 2-dimensional matrices. The arity of 3 could come from the fact that you have two sorts for the indices and one sort for the contents of the matrix.

For functions, the :funs category allows you to declare functions of the theory, the general form is:

:funs $((f_1 s_1 l, \ldots, s_k l : a), \ldots, (f_n s_1 j, \ldots, s_d j : b))$

OR

:funs $(par(X_1, \ldots, X_m)(f_1 s_1 l, \ldots, s_k l : a), \ldots, (f_n s_1 j, \ldots, s_d j : b))$

For the first version f_i represents the identifier of the function and the s_i are the sorts that represent the rank of the function meaning f_1 takes terms of sort $s_1 l, \ldots, s_(k-1)l$ and outputs a term of sort $s_k l$, :b represents an attribute of the function, this can be a predetermined attribute that helps the solver interpret how the function works or just a user defined attribute that is just used for the user to understand the function or for viewers to understand the function, user defined attributes are similar to a small comment near a function in a program. For the second version $par(X_1, \ldots, X_m)$ represent arbitrary sorts that are parameters of the function, so for example in the Core theory there is a function called = which is meant to represent equality (although it does not need to mean that since the declarations just defines the rank of the function). That function in the Core theory is defined as: $(par(A)(= \; A \;\; A \;\; Bool) : chainable)$

Here A represents an arbitrary sort and $(= A\, A\, Bool)$ means that = function takes two terms of sort A and outputs a term of sort Bool. In the instance of Core where the interpretation of the equality function is the usual one, then the equality function would take two terms of some arbitrary sort and output true if the terms are "equal" and false if they are not "equal", Of course what does it mean for two terms to be equal would have to be defined.

One interesting part of function declarations is that constants are also declared as functions with a rank that contains only one sort, meaning they are functions without inputs. For example in the theory Core true and false are functions with no inputs of rank Bool and sort Bool.

The other categories of function declarations serve the purpose of comments, to help users understand the theory being declared. These categories take strings which can be any string but is mainly there for strings representing some kind of message for a person looking at the theory declaration that explains various properties of the theory

Logic Declarations

SMT-LIB also allows the declaration of sub-logics which also restrict the models under consideration, like theories they include 5 categories to define attributes of the sub-logic which include:

- :theories
- :language
- :extensions
- :notes
- :values

The only category with predefined semantics in logic declarations is the :theories category, this defines a set of theory instances which are combined to provide the logic with a combined theory which restricts the models of the logic to those that interpret in the same way as the combined theory. So logics are fully defined by the sort symbols and function symbols of the theories that form the logic. For example The Core theory can be used to give rise to propositional logic, by adding infinitely many functions with no arguments of sort Bool representing boolean atomic formulas and give the propositional logic interpretation to its function symbols. This is the case since you would have the alphabet (boolean variables and logical connectives), the necessary logical connectives, and well formed formula rules defined by the ranks of the functions and by SMT-LIB itself. SMT-LIB also allows the formation of higher order logics by way of more complex theories.

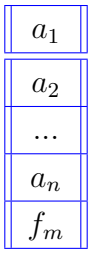
The other attributes are mainly user-defined and are used for the purpose of helping onself or others understand the logic, in addition to helping others buildings SMT solvers that use the SMT-LIB standard to more easily accommodate a specific logic into their solvers.

5.1.4 Semantics

The semantics of SMT-LIB defines what the different commands that a user inputs do. In other words the meaning of the commands in the SMT-LIB format. Many commands are available to the user and how these commands affect the state of the SMT solver can be represented using a graph. It can be represented using 4 states, the start mode state, the assert mode state, the sat mode state, and the unsat mode state. The start mode is the state of the solver before a signature is given with a corresponding interpretation for that signature that will define various functions and sort symbols. After an interpretation is given and a logic is set the solver proceeds into the assert state. You need to set a logic to give specific interpretations to certain function symbols. For example, z3 gives + and * their usual interpretation in mathematics. The assert state is where the solver can interpret and accept assertions, declarations, and definitions to check whether the assertions are satisfiable (True in some models where the background theory is true). If the assertions are satisfiable it goes to the sat mode where you can get things like a model where the assertions are true, if the assertions are not satisfiable it goes to the unsat mode where you can get for example a proof that it is unsatisfiable.

There are many commands one can pass as input for the solver to evaluate. There are assert commands, which are the commands that assert statements and are evaluated by the solver to see if the conjunction of assertions is satisfiable or not. There are also declare-const commands which are used to declare constant symbols, as stated before they are treated as functions of no arguments by the SMT-LIB standard. There are also declare-fun commands which are used to declare functions with their sort, without defining them so for example it allows to declare a function from Int to Int. There are also define-fun which allows one to define a specific function such that it tells you what output to give given different inputs. There is also a declare-datatype command which allows you to create algebraic datatypes and build types from more primitive types. There is the check-sat command which checks the satisfiability of the assert statements and outputs sat or unsat. There is also the get-model command which provides a model in which the assert statements are satisfiable. For the purposes of our project we believe we will have to use define-fun commands a lot since we want to encode our quantifiers as SMT-LIB functions.

According to the SMT-LIB standard, a solver which is compliant with the standard would possess an assertion stack that keeps track of the assertions being made. In fact, z3 which is one of the solvers used by OpenJML does just that. The (check-sat) command checks whether the assertions currently in the stack are all satisfiable or definable if taken as a conjunction, meaning if in the stack there are the following assertions:



the solver would check whether the statement:

$$a_1 \wedge a_2 \wedge \ldots \wedge a_n$$

is a satisfiable statement. and the function f_m is definable

the SMT-LIB standard allows you to pop and push assertions and declarations out of the stack so that it can only check the satisfiability of certain assertions in the assertion stack and define only certain declarations or consider only certain definitions.

Of particular interest to this project is the fact that function definitions are semantically equivalent to function declarations followed by a certain type of assertion about that function. More specifically the two following are equivalent:

```
(define-fun f ((x1 s1) ... (xn sn)) s t)
```

is equivalent to

```
(declare-fun f ((x1 s1) ... (xn sn)) s)

(assert (forall ((x1 s1) ... (xn sn)) (= (f x1 ... xn) t))
```

The reason this is of particular interest is because we want to encode our quantifiers as SMT-LIB functions so understanding very well how they work will help us to form an appropriate encoding.

5.2 SMT Solvers

In general users will use the Z3 SMT solver as it is OpenJML's default; however, certain scenarios may lead to alternative solvers due to the advantages others offer. The SMT-LIB specification allows drop-in replacement for Z3. In fact, the SMT-COMP hosted every year

serves to promote development of SMT solvers and their usage [8]. They use SMT-LIB for all benchmarks to ensure an equal playing field. This competition serves as the de-facto test of a solver to see how it stacks up with alternatives in terms of coverage and speed.

The SMT-COMP compares many of SMT solvers against a suite of benchmarks for accuracy and speed. SMT-COMP includes 157 divisions in 4 different tracks. Each division includes a bunch of benchmarks in SMT-LIB format to test the solvers with. The purpose is to highlight a specific feature of SMT-LIB or replicate a real world use case. In 2020, there were 77 different solvers:

AProVE, Bitwuzla, COLIBRI, OpenSMT, SMT-RAT-MCSAT, SMT-RAT, SMTInterpol-remus, SMTInterpol, SMTS cube-and-conquer, SMTS portfolio, STP-CMS-Cloud, STP-parallel, STP, UltimateEliminator+MathSAT, Vampire, Yices2-QS, Yices2, Yices2 incremental, Yices2 model-validatio, YicesLS, Z3str4, cvc5-gg, cvc5-inc, cvc5-mv, cvc5-uc, cvc5, iProver, mc2, veriT+raSAT+Redlog, veriT, 2018-CVC4, 2018-MathSAT (incremental), 2018-Yices, 2018-Yices (incremental), 2018-Z3, 2018-Z3 (incremental), 2019-CVC4-inc, 2019-CVC4, 2019-MathSAT-default, 2019-Par4, 2019-SMTInterpol, 2019-Yices 2.6.2, 2019-Yices 2.6.2 Incremental, 2019-Z3, 2020-Bitwuzla-fixed, 2020-Bitwuzla, 2020-CVC4-inc, 2020-CVC4-uc, 2020-CVC4, 2020-MathSAT5, 2020-OpenSMT, 2020-SMT-RAT, 2020-SMTInterpol-fixed, 2020-Vampire, 2020-Yices2-fixed, 2020-Yices2-fixed Model Validation, 2020-Yices2-fixed incremental, 2020-Yices2, 2020-Yices2 Model Validation, 2020-Yices2 incremental, 2020-z3, Bitwuzla - fixed, COLIBRI - fixed, MathSAT5, OpenSMT - fixed, Par4, Vampire - fixed, Z3str4 - fixed, cvc5-inc - fixed, cvc5-mv - fixed, cvc5-uc - fixed, cvc5 - fixed, iProver - fixed, iProver - fixed2, z3-mv, and z3n.

Z3 is one of the fasted during the 2020 competition; however, it comes at the cost at being unable to verify all programs. For instance, factorials cannot be verified in Z3 because multiplications by non-constants is not supported. Solvers tend to be optimized for different tasks and thus compete in different tracks. One particularly interesting track is the parallel one. SMT Solvers that take advantage of multi-core processing may prove to be significantly faster than others.

5.2.1 Simplify

Simplify is a SMT solver in [5] and they demonstrate it is more feature complete than Z3. This solver is highlighted due to its direct use in the C# implementation of this project. It was also used in the original ESC/Java program checker.

5.2.2 Cooperating Validity Checker

Another SMT solver bundled with OpenJML is Cooperating Validity Checker (CVC4). This is an open-source automatic theorem prover that proves the validity of first-order formulas [7]. Unlike Z3, CVC4 is academically supported as a joint project of Stanford University and The University of Iowa. The primary goal is to an open and extensible SMT engine to be used either as a stand-alone tool or as an library. At SMT-COMP 2020, CVC4 received 89 1st places.

5.2.3 Z3 Solver

Z3 is an open source theorem prover from Microsoft Research [9]. Z3 is associated with many publications and used in a variety of applications. It includes bindings for various languages primarily: C, C++, Java, and Python. The SMT-LIB solving standard is the default input format. OpenJML ships with Z3 making it the standard choice for SMT solving; however, it is conceivable a user may use an alternative solver which is why the SMT-LIB standard will be used.

Theorem provers have used a brute-force search approach to solve expressions. Pruning branches is key to efficient solving. Z3 uses a current candidate model to limit the search. Instead of relying on evaluating all possible values, the method asks the current candidate for what values it already assumes.

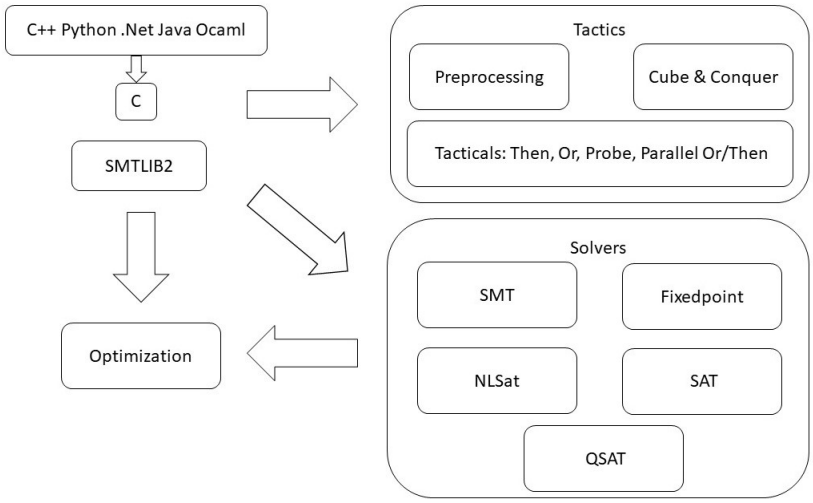


Figure 4: Z3 Processing [10]

Z3 will perform some simplification where possible and then use various theories to solve the expressions (Figure 4). When expression are large, eliminating variables and simplifying is important for low memory use and faster run times. Z3 is also incremental and will reuse lemmas to short cut solving.

Example of Z3 python statements converted to SMT-LIB syntax.

```
s = Solver()

x = Int('x') # (declare-fun x () Int)
y = Int('y') # (declare-fun y () Int)

# (assert
#  (let (($x8 (= x 2)))
#  (and $x8)))
s.add(x == 2)

s.check() # (check-sat)
s.model() # (get-model)
```

Because Python allows for better introspection and debugging than pure SMT-LIB statements, z3py will be used during development. In addition, Python solver expressions can be converted to SMT-LIB statements quite easily:

```
def solver_to_smt(f):
    f = And(*f.assertions())
    return Z3_benchmark_to_smtlib_string(f.ctx_ref(), "", "", "unknown", "", 0,
        (Ast * 0)(), f.as_ast())
```

The z3py translations tend to be more verbose than pure SMT statements, so these outputs will not be directly used. Consider trying to prove $x \in \mathbb{Z}$ can equal 2 (i.e. $x = 2$). The Z3 translation can be see above. The SMT-LIB version is as follows:

```
(declare-fun x () Int)
(assert (= x 2))
(check-sat)
(get-model)
```

As one can see in the axioms, the forall operator is critical to verifying the generalized quantifiers.

```
s = Solver()
f = Function('f', IntSort(), IntSort(), IntSort())
x, y, a, b = Ints('x y a b')

# defines f to return the sum of x and y
s.add(ForAll([x, y], f(x, y) == x+y))

# ensure a and b are positive, and find f(a, b) == 7
s.add(a > 0)
s.add(b > 0)
s.add(f(a, b) == 7)
```

In some cases proving statements for all x can be impossible or too time consuming, so pattern matching or 'triggers' are used as short cuts. These patterns give the SMT solver more information about some function in order to prove generalize functions.

5.2.4 Handling Results

Program verification attempts to prove the program is incorrect by finding a counter example. For instance, consider trying to prove an add1 function. If $\text{add1}(x) \neq x + 1$ is satisfiable for some x, add1 would be incorrect. The SMT-LIB specifics how SMT Solvers should display their results. This increases the replaceable of the tools and allows users to easily upgrade to other versions that might be more efficient for their use case.

When verifying programs, there are three main options:

- "unsat" - the program is correct as no counter examples were found.
- "sat" - the program is incorrect as a counter example was found.
- "unknown" - the program is unable to be verified due to some SMT solving issue (such as timeout).

Getting "sat" is the key case to discuss as the user must be notified and understand what the error is. The '(get-model)' command will return an example that satisfied the statements.

```
(model
  (define-fun x () Int
    2)
)
```

Simply printing the example to the user will not be helpful as an entire translation process occurs between the JML statements and program verification. This means the user will likely have no idea what the counter example truly refers to. Therefore, a mapping between JML statements and SMT-LIB statements is required to backtrack where the actual issue lies. Currently, OpenJML just prints the counter example model from the SMT solver,

In most cases the counter-example will not be as useful as demonstrating which line of code there is an issue with. The priority in error reporting will be highlighting the piece of code and the corresponding JML statement it is at odds with. Both statements are needed because either the implementation or the specification could be incorrect.

The current design of OpenJML is to report statements that are either wrong or unprovable, so a "correct" program should report nothing.

Consider a simple function that should always return 5. If there is a typo and the code actually returns 4, OpenJML will report the following error. There are 2 lines that error. First is the line of code the prover is unsure about and the second is the associated JML statement. This error alerts the coder of the two places that there might be an issue with. The code might contain a bug, or the specification needs to be more exact.

```
public class Example {
    //@ ensures (5 == \result);
    public static int foo() {
        return 4;
    }
}

/*
$ openjml -esc Example.java
Example.java:4: warning: The prover cannot establish an assertion
    (Postcondition: Example.java:2: ) in method foo
        return 4;
        ^
```

```
Example.java:2: warning: Associated declaration: Example.java:4:
    //@ ensures (5 == \result);
        ^
2 warnings
*/
```

In the SMT Solver the result is unknown so foo's method assertions are not valid. In this case there is no specific counter example given, except that the statements were not true. OpenJML goes up the stack to figure out where the issue is.

This example is also interesting due to its seemly simple verification. The translated SMT-LIB statements is over 200 lines from the simple 6 line program. This demonstrates that robust verification is complex and hard to do by hand which is why tools like OpenJML are so important.

Some other notes on SMT solving:

Multiple functions may contain errors and as such every function should be reported. OpenJML is unique in that each method is isolated during proving, so any dependencies will not error even if the parent is faulty. This is achieved by treating the specification as fact whenever the function is used. This means as long as the specification requirements are met, the use case should not fail regardless of the correctness of the function. This helps reduce the error reporting bloat that can be difficult to understand.

Some errors will not be "verification" related, such as parsing errors. These should be reported early and often before SMT solving begins as solving takes sufficiently long. Most of these type of errors will be handled by the OpenJDK parser; however, if errors occur during the SMT translation the user needs to be notified before trying to perform SMT solving.

Tests will be in place to ensure translated SMT statements are always valid; however, in a debugging mode (such as -verbose), SMT errors should be logged so the OpenJML developers can understand why a translation might have failed.

5.3 OpenJML

5.4 Run-time Assertion Checking

5.4.1 Introduction to Run-time Assertion Checking

OpenJML allows for the use of JML annotations to modify compiled Java byte code. This modified byte code can then be run on Java virtual machine with some additional features to find and warn the user about errors within code at run-time. For example we might want to verify that the below method will receive valid inputs.

```
public static int addArrElem(int i, int j, int[] arr) {
    return arr[i] + arr[j];
}
```

This method simply takes two indices and an array and sums the values at the given locations. We know that there are few simple restrictions we can put on the functions parameters to prevent errors. An array can not be indexed using a negative number in Java, so we know that our values for i and j must be positive. Similarly, we know that the indices must not extend outside of the array. To specify this using JML we might add the following annotations.

```
//@ requires i >= 0 && j >=0;
//@ requires arr.length > i && arr.length > j;
```

These annotations specify to the run-time assertion checker the same statements that they specify to us. This means, if we run this code though the run-time assertion checker, it will warn us in the case of these assertions being false. Lets examine the output of the following program.

```
public static void main(String[] args) {
       int[] arr = {1,2,3,4};
       System.out.println(addArrElem(1, -2, arr));
}

//@ requires i >= 0 && j >=0;
```

```
//@ requires arr.length > i && arr.length > j;
public static int addArrElem(int i, int j, int[] arr) {
    return arr[i] + arr[j];
}
```

this program compiles but the JML run-time assertion checker catches the exception and warns the user to look at the first assertion in the program above.

5.4.2 How Run-time Assertion Checking Works In OpenJML

OpenJML takes a compilation-based approach to run-time assertion checking. This means that using a specific compiler the JML annotations are compiled into traditional Java code and byte-code. This means the assertions are directly converted into java code that you could insert and run in a program by itself [11]. Thus, a user could compile the code with a traditional javac java compiler and run it without any overhead of the JML run time assertion checking additional code.

Given that JML takes a compilation-based approach, there needs to be a compiler that can convert the JML annotations into Java code and byte code. The JMLc compiler takes three additional compilation passes over javac to type check, create the java methods, and hook everything together [11]. The compiled Java byte-code is then able to be run on the Java Virtual Machine with a classpath including the JML run-time environment.

5.4.3 JML Interface With The Java Virtual Machine

As we demonstrated in the previous section, code compiled with the run-time assertion checker can then be run though the Java Virtual Machine to produce warnings specific to the annotations added written. The interface with the virtual machine is similar to most java programs.

The Java program including verifications is first passed to OpenJML compiler using a command such as

bash: java -jar $OpenJML/openjml.jar -rac <main program name>.java

At this stage you will now have a a compiled class file which has the inserted code for the

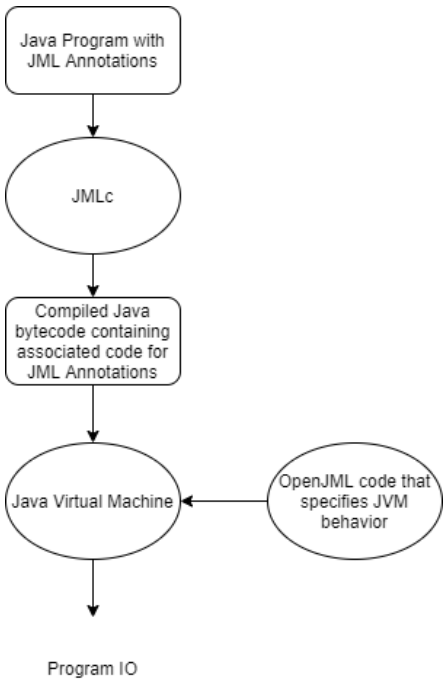


Figure 5: RAC compilation and run process

annotations as described in the previous section. To run the code you can execute

bash: java -cp ".;$OpenJML/jmlruntime.jar <main program name>

This step will run the compiled byte code on the Java Virtual Machine with a class path pointing to the JML run time. The code runs as it would on a normal JVM and produces outputs pertaining the the assertions.

It is important to understand the -cp tag. With this, we are specifying the custom class path from OpenJML. This allows for the virtual machine to react with some custom behavior given arguments.

5.5 Extended Static Checking

OpenJML's extended static checker performs two major tasks: compilation and running. The extended static checker compiles and runs a Java class to see if there exist any errors. The primary difference between this process and the standard Java compiler is that instead of compiling to Java bytecode to run in the Java Virtual Machine, OpenJML compiles to

SMT-LIB. Similar to a compiler, the extended static checker contains a lexer, parser, and code generator.

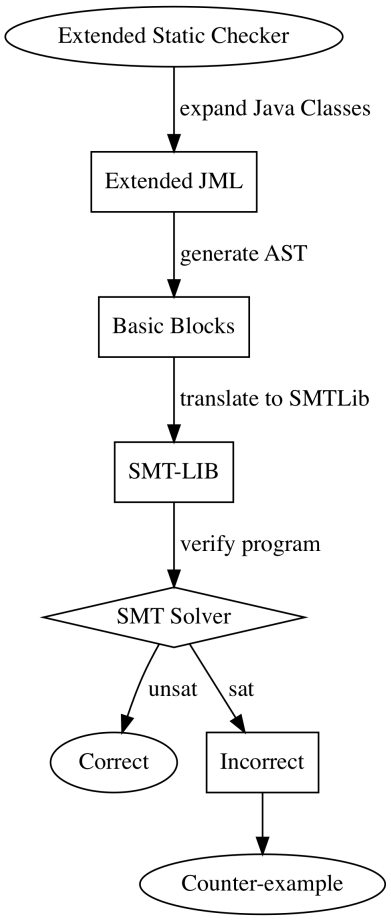


Figure 6: OpenJML's Extended Static Checker Flow

Practically speaking, OpenJML first generates an Extended Java form with added verbiage to encapsulate implicit Java conditions. Second, the new class is parsed using an extended version of OpenJDK. The result is an Abstract Syntax Tree in the form of basic blocks. Finally, the Abstract Syntax Tree is translated to SMT-LIB. This entire process is essentially transpiling Java to SMT-LIB. Finally, OpenJML runs an SMT Solver to determine the validity of the code. The entire process is streamlined into a single command that will compile and run the resulting SMT-LIB (Figure 6).

Each phase of this process is viewable with the '–verbose' flag. Although it appears that multiple text files are created, they simply are a pretty-printed version of an internal data structure. The extended static checker is particularly interesting due to its unique conditions. It receives no real runtime data and has "infinite' time to run. There are various flags that

will configure how the checker should operate. For example, one can set the time limit for SMT solving or can receive the SMT-LIB model of a counterexample. These options are useful tools for someone trying to better understand why their code might not be verifying as intended.

6 Generalized Quantifiers

6.1 Goals of Quantifiers in OpenJML

The goals of the quantifiers in OpenJML are as follows:

- Verify for all written test cases successfully
- Return detailed error reporting on improper specifications
- Return detailed error reporting on unsatisfiable specifications
- Sufficiently comment any written code for easy maintenance.

Generalized quantifiers follow this specification:

```
spec-quantified-expr ::= ( quantifier quantified-var-decls ;
                         [ [ predicate ] ; ]
                         spec-expression )
quantifier ::= \forall | \exists
       | \max | \min | \num_of | \product | \sum
quantified-var-decls ::= [ bound-var-modifiers ] type-spec
    quantified-var-declarator
                       [ , quantified-var-declarator ] ...
quantified-var-declarator ::= ident [ dims ]
spec-variable-declarators ::= spec-variable-declarator
                        [ , spec-variable-declarator ] ...
spec-variable-declarator ::= ident [ dims ]
                           [ = spec-initializer ]
spec-array-initializer ::= { [ spec-initializer
            [ , spec-initializer ] ... [ , ] ] }
spec-initializer ::= spec-expression
       | spec-array-initializer
```

```
bound-var-modifiers ::= non_null | nullable
```

The parsing of these expressions is already handled in OpenJML, so this project will focus on utilizing the abstract syntax tree produced from the grammar above.

In general, a quantifier expression looks like:

$$(\text{QUANTIFIER } T\ x;\ R(x);\ P(x))$$

Well definedness

Each expression has rules for when it is well-defined (see the reference manual). Assuming that the expression is syntactically and semantically valid, then if it is not well-defined it is ill-defined.

All ill-defined expressions should generate warnings during static verification and should either not compile or throw an exception during the runtime assertion checker.

$R(x)$ must be well-defined for all possible x of type T and $P(x)$ must be well-defined for all x when $R(x)$ holds.

The quantifiers \max, \min, \product, \sum, and \num_of are generalized quantifiers that return the maximum, minimum, product, sum, or number of the values of the expressions given. The expression body, $P(x)$ must only be a numeric type (int or double for instance). Any $P(x)$ that returns a non-numeric type would result in an error at compile time.

OpenJML has limited support for these quantifiers:

- Expressions are parsed into an abstract syntax tree.
- Runtime assertion checking limited functionality for certain Rs. The main limitation is with the range that can be iterated over.
- Extended static checking is not supported at all.

6.2 \sum

6.2.1 Introduction

The \sum quantifier will be used to add together elements equaling the expression for each iteration within the given range

```
(\sum int i; 0 <= i && i < 5; i);
```

The example above is a simple quantifier comprehension using the \sum expression. Let us break down this expression to better understand what it will do. First, the *int i* signifies that our quantified variable will be of type integer. This provides an important restriction on this variable and the output of the whole expression (we will get to this later). Next is the range that the quantified expression will be evaluated upon. Unlike many traditional programming languages, you will not normally specify an ordering to the execution of these iterations. In this example, i must be between 0 (inclusive) and 5 (exclusive). Lastly is the quantified expression, i. This quantified expression will be evaluated at each iteration and added to the cumulative sum of the whole expression. So how will this evaluate?

1. When $i = 0$ then $0 <= 0$ is true, and $0 < 5$ is true so the current total is 0
2. When $i = 1$ then $0 <= 1$ is true, and $1 < 5$ is true so the current total is $1 + 0$
3. When $i = 2$ then $0 <= 2$ is true, and $2 < 5$ is true so the current total is $2 + 1 + 0$
4. When $i = 3$ then $0 <= 3$ is true, and $3 < 5$ is true so the current total is $3 + 2 + 1 + 0$
5. When $i = 4$ then $0 <= 4$ is true, and $1 < 4$ is true so the current total is $4+3+2+1+0$
6. When $i = 5$ then $0 <= 5$ is true, and $5 < 5$ is false so the current total is $4+3+2+1+0$

This means that the expression should evaluate to 10. Now this manual evaluation might be misleading to the computational evaluation of the expression. As previously noted, there is no specified ordering to the execution in the expression. This means that the solver will attempt to use every possible value in the expression. Given that we specified that the quantified variable is an integer, the solver will try every value that satisfies the conditions of being an integer within Java. So on the iteration where $i = -291,239,292$, i will not satisfy the range expression and will not be counted within the sum.

At this point, we understand that the sum quantifier calculates a numerical value equaling the sum of the quantified expression at each iteration possible within the data type that satisfies the range expression. By tweaking the variable, range, and quantified expression, we can represent many different problems.

The quantified variable can be any of Java's primitive numerical data types. The two different types of numbers, floating-point versus whole numbers, act differently in many cases. Important to the \sum quantifier, integer data types overflow and underflow wrapping around from the max to minimum value and vice versa. Differently, the floating-point data types contain numbers known as positive and negative infinity which represent a number larger or smaller than the maximum or minimum value that data type can contain respectively.

The range expression is a boolean that determines what values should be evaluated within the quantified expression.

```
(\sum int i; i == 2 && i == 10000; i);
```

The example above would return 10002 since the quantified variable was expressed at only 2 and 10000. This range expression can be any boolean leading to many interesting edge cases and possibilities.

The sum quantifier will also be able to iterate over elements of arrays. For example

```
(\sum int i; i >= 0 && i < arr.length; arr[i]);
```

This example above represents the sum of all elements with some array arr. Interestingly, the data type of the array elements does not have to be integer as the quantified variable is specified. However, the data type still needs to be of a Java primitive numerical type.

6.2.2 Example

Many programmers find themselves summing up elements in an array. Much of the time this could be part of a more complex calculation or the sum could be adding together elements and computing intermediary calculations. However, we will simply be summing all the elements in an array together to both gain understanding of how the quantifier works, and to provide a good base level test case for development.

```
//@ ensures \result == (\sum int i; 0 <= i < arr.length; arr[i]);
public static int sumArrayElem(int[] arr) {
    int sum = 0;

    //@ maintaining sum == (\sum int j; 0 <= j < i; arr[j]);
    for(int i=0; i<arr.length; i++) {
        sum += arr[i];
    }
    return sum;
}
```

There are two quantifier comprehensions in this example method. First of which is the ensures result statement which is checking the postcondition. Specifically, It checks that the return value of the method equals the sum of array element integer values from the input array. This specification defines to any developer reading the code, that this method is meant to be summing array elements of the entire array. This provides both use in the design by contract application of JML, as well as the checking of program correctness. Within the method, there is a loop variant. This expression strictly states that all elements up the index i must sum to the current value of sum. A loop variant is evaluated for loop iteration meaning that the checking will ensure that at each possible iteration the value within the variable sum will be correct.

Importantly, We expect the above test case to verify. This means that for every possible input that a user could input, the output would match the output of the method and the sum variable would be equal to the inner quantifier's value. Just as important to testing software as test cases that work, are test cases that do not work.

```
//@ ensures \result == (\sum int i; 0 <= i < arr.length; arr[i]);
public static int sumArrayElem(int[] arr) {
    int sum = 0;

    //@ maintaining sum == (\sum int j; 0 <= j < i; arr[j]);
    for(int i=0; i<arr.length; i++) {
        sum += arr[i];
    }
    return sum+1;
}
```

With a change as subtle as adding one to the return value, we expect the program will not verify now. It is worth noting that there is still nothing invalid about the loop variant, but the postconditions check would no longer match the actual output.

Similar to the above test cases it will be integral to the success of the project to define cases that are both satisfiable and unsatisfiable. Once the simple test cases are working we can then experiment with some more complex realistic use cases.

6.2.3 Example Use-cases

By providing possible use cases, we will both be creating test cases that will help in the development of the patch to OpenJML, and defining some cases where developers might find the \sum quantifier useful.

Sums are one of the most fundamental concepts of computer science. In the very early stages of learning a computer science one might learn some concepts of summations and how they can be simplified into non quantified expressions.

$$\Sigma_{i=0}^{n} i = 1 + 2 + 3 + ... + n = \frac{n(n+1)}{2}$$

Sure, this looks to work when we do some tests by hand on our calculator. However, what if we want to see that this holds for all integers?

```
//@ ensures \result == (\sum int i; i>0 && i<=n; i);
public static int sumI(int n) {
    return (n * (n-1))/ 2;
}
```

This leads to the concepts of using series for approximation. There are many uses of summations within mathematics where the computation of the series might take many iterations before an accurate solution is found. If you have ever investigated Taylor series, Maclaurin series, or power series you might know that they can be used to approximate many calculations.

Sums are also common in many fundamental computer science data structures and algorithms. For example a cumulative frequency array is a data structure used to find the subtotal of all elements up to the current element within an array.

For the array [1,2,3,4,5] the cumulative frequency is [1,3,6,10,15]

These can be useful in many mathematical algorithms. For example say you needed to find the sum of a list of elements, you would use a sum and calculate a total. Now lets change this a bit, and say you wanted to know the sum of the first five elements, but also the sum of the first ten. By calculating that sum iterative to the fifth element, then again but to the tenth we have slow repeated calculations. To solve this, we can sum the elements up to and including the current element by adding the previous number to the current number and storing it in the location of the current number. Now, to get the fifth and tenth elements sums, you can just index into the array at those two locations. Interestingly, we can also subtract the value at the fifth element from the tenth, and now we have the sum of elements in the array between index five and ten. Clearly these cumulative frequency arrays can be applied in many places, but lets just examine a simple example.

```
//@ ensures \result == ((\sum int j; 0<=j<high; arr[j]) - (\sum int i; 0<=i<low;
    arr[i]));
public static int cumFreq(int[] arr, int low, int high) {

    //@ maintaining arr[i] == (\sum int k; 0<=k<i ; arr[k]) ;
    for(int i=1; i<n; i++) {
        arr[i]+=arr[i-1];
    }

    return arr[high] - arr[low];
}
```

Much of Java web development requires classes that represent a database schema. Say there is a database which is defined by the following schema

Column	Type
FirstName	String
LastName	String
NumDogs	Integer
NumCats	Integer

There could be a request to create an endpoint that finds the total number of pets owned by people named Bob. We could go about this by querying by first name and storing all

the elements within an array. Then we would sum the dogs and cats into the total sum over each element in our array.

```
class petOwner {
    private String firstName;
    private String lastName;
    private int numDogs;
    private int numCats;

    // Assume this class has getters, setters, constructor...
}
```

With the class defined to fit the schema of the database we could query data into a list or array of petOwner elements. Once we have constructed this data structure, we would complete a summation to check the number of pets owned.

```
private int getPetCounts() {
    // Gather data
    ArrayList<petOwner> owners = new ArrayList<>(getAllOwnersByName("Bob"));

    // Sum owners
    int pets = 0;
    for(petOwner owner: owners) {
        pets += owner.getuNumDogs() + owner.getNumCats();
    }
    return pets;
}
```

The above calculations could be verified using a sum quantifier comprehension even though its a list object holdings our own custom class. This might look like

```
(\sum; i>=0 && i < owners.size();
     owners.get(i).getuNumDogs() + owners.get(i).getNumCats());
```

Yes, you can do this in SQL with a group by expression. However, that code would not be able to be verified by OpenJML. This is more of a simple example than anything, but could be useful given that you are not querying the whole data set and filtering it from memory.

6.2.4 Edge Cases

Similarly important to finding use cases we also need to find some edge cases that could challenge the bounds of the \sum quantifier. Once we have the software working effectively on our simple test cases and more realistic use cases, we will need to verify that as many possible edge cases are covered as possible.

Some of the topics worth considering in edge case testing will be:

1. Covering the basic edge cases provided in the JML manual, including true and false range predicates.
2. Verifying effectiveness with floating point numbers by handling infinities and NaN
3. Checking that we can handle problems with whole number datatype including overflows and underflows
4. Pushing the limits of the OpenJML math modes by making sure that we comply as expected

First off and most simply, we know the range is just a boolean expression. Given this, what happens if the boolean is always true, or always false? For always true we assume that the solver would just test will every valid number within the data type defined for the quantified variable. On the other hand, always false should strictly return zero for \sum according the JML reference manual. We must follow this specification, so we should create a test case to verify we have this covered.

```
//@ ensures \result == (\sum int i; false; i);
public static double AlwaysFalseRange() {
    int sum = 0;
    //@ maintaining sum == (\sum short j; false ; j);
    for(long i=1; i<0; i++) {
        sum++;
    }
    return sum;
}
```

In this case we have a method with a loop that never iterates. This means the sum should remain at zero, and the loop variant should be satisfied. With this, we know that the return

sum will also be zero which should also match, verifying the method. Since each other case we use will have a non-false range, we will use them as the opposite example to prove this case is working correctly.

Next lets create a case that will overflowing a whole number. We will use short here to keep the numbers more manageable.

```
//@ ensures \result == (\sum short i; 0<=i<high; i*i);
public static int sumSquares(int high) {
    short sum = 0;

    //@ maintaining sum == (\sum short j; 0<=j<i ; j*j);
    for(short i=0; i<n; i++) {
        sum+=(i*i);
    }
    return sum;
}
```

As we know, this case should quickly overflow the short max value in Java of 32767. Given this, we can assume that this would not verify, as the solver should recognize the possibility of overflow on all inputs possible.

Importantly, in order to get a true test, we should also try to get a similar function to verify. By restricting the input to a specific size, the solver will know that we will only pass shorts that will be within the valid range for shorts before an overflow.

```
//@ requires 0 <=high <10;
//@ ensures \result == (\sum short i; 0 <= i < high; i*i);
public static int sumSquares(int high) {
    short sum = 0;

    //@ maintaining sum == (\sum short j; 0<=j<i ; j*j);
    for(short i=0; i<n; i++) {
        sum+=(i*i);
    }
    return sum;
}
```

By adding the above line above the function declaration we restrict the input enough to not overflow the short data type. At this point we would expect for our code to return satisfiable for the first example, and un-satisfiable once we add the requires line.

As we move into floating point numbers, some of the edge cases get more complex. We will not be considering cases of lack of floating point precision in this example, since we assume the user should be using a comparator other than equals, and using a large enough delta. However, the concepts of Not a Number (NaN) and infinities are interesting cases including how they interact and relate to other numbers.

We should have a test case to cover some of the issues commonly seen with floating point numbers. Given that we already covered an integer max value case, we should proceed with a NaN case over an infinity case. So how is JML supposed to deal with NaN, and how does that compare to a sum in Java. They should both act the same, that is, in any interaction with NaN the return value should be NaN. For example, if we add NaN to one, the result should be NaN. Now lets take a look at a JML test case.

```
public static void main(String[] args) {
    double[] arr = {1.2, 2.2, 4.0, Double.NaN, 2.0};
    SumDoubles(arr);
}

//@ ensures \result == (\sum int i; 0<=i<high; arr[i]);
public static double SumDoubles(double[] arr) {
    double sum = 0f;

    //@ maintaining sum == (\sum int j; 0<=j<i ; arr[j]);
    for(int i=0; i<arr.length; i++) {
        sum+=arr[i];
    }

    return sum;
}
```

This case is very similar to our simple example test case. However, we have shifted the datatype to double on the array. Importantly, the quantified variable in this case is still an integer. In this case we have included the array in the main function. We would expect this case to verify, since even though we know that the NaN case is possible, both sides will

expect the NaN to appear.

Now, you might be wondering what happens when you create a quantifier comprehension to iterate over a floating point. Given the language specification the quantified variable can be of any type, including a floating point type. We must create test cases that verify that we provide the correct warning if you cannot quantify over a given range. For example, let look at two possibilities.

```
(\sum double i; 0 < i <= 1; i)
(\sum double i; i % 0.5 == 0 && 0 < i <= 1; i)
```

In the above example, line one should provide a warning to the user. Given that there is an unaccountably infinite amount of numbers between 0 and 1, and JML assumes the calculations are done in non-type specific math, there is an infinite amount of loops that this would take to calculate. For line two, since there are only two values that meet the range criteria, we should not produce a warning. This expression will evaluate to $1.5f$.

We know that the JML executes mathematical computations on one of a few different math modes. However, when not in the Java specific math mode, we expect that the intermediary values can be larger or smaller than the max and min value of the quantified variable data type. This leads the the question of what should happen when you quantify below the given array.

```
int[] arr = {2, Integer.MAX_VALUE, Integer.MAX_VALUE,
            Integer.MAX_VALUE, Integer.MIN_VALUE };
```

Summing this with the method sumArrayElem method should give the same results on different math modes given the way that Java converts data types. If we compute this with both standards, the result should be negative one. Given the Java math mode, we would see an overflow and then an underflow as well. On the IEEE mode, we would not see this happen. However, when converting the data from a larger number to an integer, we would expect it to wrap around from zero to min value and end up at negative one.

```
Java Mode : {2, -2147483647, 0, 2147483647, -1}
IEEE Mode : {2, 2147483649, 4294967296, 6442450943, 4294967295}
```

In order to push the limits of these type conversions, we will have to use floating point

numbers. Lets examine the function below.

```
//@ ensures \result == (\sum int i; 0 <= i < arr.length; arr[i]);
public static float sumFloatElem(float[] arr) {
    float sum = 0;

    //@ maintaining sum == (\sum int j; 0 <= j < i; arr[j]);
    for(int i=0; i<arr.length; i++) {
        sum += arr[i];
    }
    return sum;
}
```

This function is similar to our sum doubles function from earlier, but importantly for this example uses the type float for the sum. Lets use this method to sum the following array elements.

```
float [] arr = {2, Float.MAX_VALUE,Float.MAX_VALUE, -Float.MAX_VALUE }
```

When summing over the following array within Java math mode we reach the double value infinity after the second max value. Importantly anything added to infinity will result in infinity. Even after adding a negative max value we remain at infinity. This contrasts the IEEE math mode which uses infinite precision. In this case, we add the second max value, then remove one of them. After removing the second, we can convert to float and not be at infinity. Interestingly, this means that we will get different values with the different math modes. It will be worth considering providing a warning if an intermediary calculations value exceeds infinity of the underlying data type. If this issue is not handled properly, we could cause problems with verifications.

```
Java Mode : {2.0, 3.4028235E38, Infinity, Infinity}
IEEE Mode : {2.0, 3.4028234663852886E38, 6.805646932770577E38, 3.4028235E38}
```

6.2.5 Requirements

The \sum quantifier must maintain a number of requirements imposed by the JML specification language. First and most obviously, it should verify the sum of a set of elements that satisfy an arbitrary satisfiable predicate. That is to say it should properly take the sum over a given array-like structure.

As it stands, SMT-Solvers do not have a neat way to express the concept of a sum. For this quantifier to work properly, our group must implement a pipeline that allows sums to be parsed into SMT and subsequently understood by SMT solvers.

6.3 \num_of

6.3.1 Introduction

The \num_of quantifier counts the number of elements within the specified range that satisfy the given Boolean expression. This can include primitive types such as integers, objects, or other types. This indeed sounds very similar to the \sum quantifier, because it is. The \num_of quantifier can be considered a simple modification to the \sum quantifier. A specification including \sum can be configured in a way to only sum one for every satisfying element with the general form:

```
(\sum T x; R(x) && P(x); 1L);
```

Internally, OpenJML will process the quantifier as above reducing it to the equivalent sum statement. Although the functionality of \num_of is already entirely covered by sum, its use lies in that it is more intuitive for the end-user. By allowing the user to be more directly expressive, their specifications become simpler and thus easier for them to understand. This is the primary use for the duplicate functionality.

Furthermore and a slight insight into implementation, a bulk of the work required in the implementation of both \sum and \num_of revolves around not how they work semantically but in the conversion from how they work on paper to how the SMT-Solver can prove them. As it stands SMT-LIB does not provide a neat way to express the concept of summation and because of this, a custom pipeline must be constructed to express this summation notation for \sum to be executed properly. Once a pipeline has been established for \sum

the implementation of \num_of is trivial.

6.3.2 Example

```
(\num_of int i; 0 <= i && i < 10; true);
```

The above example is a basic expression making use of the \num_of quantifier. Deconstructing this statement we have

1. We specify the type of the quantifier variable, in this case, we are specifying that it is an integer
2. We provide the range over which to evaluate the quantified expression. In this particular case i must exist between $[0, 10)$
3. We then specify the predicate or the boolean function which we will map across the range of values. In this case, we are mapping a simple 'True' which will return true for all elements over the range. For each true, we will add 1 to the number of elements

We would therefore expect this to evaluate to 10. The process by which we do so is illustrated in the table below:

											SUM
Iteration	0	1	2	3	4	5	6	7	8	9	
Evaluation	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
Val to Sum	1	1	1	1	1	1	1	1	1	1	10

Constructing the quantifier in this way allows us to cover every intended use case for the \num_of quantifier. The \num_of quantifier can also easily be understood using sigma notation. Consider the following equation, which is exactly equivalent to Example (2) above:

$$\sum_{n=0}^{9} 1 = 10$$

For any arbitrary case, we can convert the \num_of quantifier as follows:

$$\sum_{n=m}^{M} 1 = (M - n) + 1$$

where

- M is the upper bound of the given range
- m is the lower bound of the given range

Although this simple conversion to sigma notation will work for simple uses of the \num_of quantifier, it notably does not make use of the Boolean predicate to evaluate when the sum should be performed. We are not constrained to explicitly defined ranges either. All quantifiers can make use of any java function which returns a primitive type. On top of this, we are not guaranteed to have a continuous range over which to map the predicate.

OpenJML currently supports any ranges that (do not have to be satisfiable but this has interesting side-effects) be satisfied. For the range of any statement that we express with integer types, we first begin with $\mathbb{Z}$ which is then subset by the range expressions one at a time. For example, suppose our specified range was ($i >= 0$ && $i <= 10$) As follows:

1. $\mathbb{Z}$
2. $S = \mathbb{Z} \cap (i >= 0)$ - Includes all integers greater then or equal to zero
3. $F = S \cap (i <= 10)$ - Includes only those integers who are between 0-10 inclusive.

Using this information we can construct a final and more appropriate sigma conversion:

$$\sum_{n=1}^{|I|}(\sum_{n=m_i}^{M_i} P(i))$$

where

- I is the set of continuous intervals
- $|I|$ is the cardinality of the set of continuous intervals. (i.e. [1,2],[300,400] == 2)
- m_i is the lower bound of the current continuous interval

- M_i is the upper bound of the current continuous interval
- $P(i)$ is the Boolean predicate being mapped over each range. For satisfying entries P(i) returns True.

6.3.3 Example Use Cases

- Manipulating data-structures where the number of satisfying elements is changed and easily calculated
    1. For example, say we were training a machine learning model but the data in which we had to train it was too large. A common solution is something called dimensionality reduction, where the number of columns or $features$ in our dataset is reduced programmatically. If we wanted to reduce the number of features by an amount n, it is fairly trivial to verify this transformation using the \num_of quantifier. We can specify this statement as follows

```
(\num_of int i; 0 <= i && i < dataset.columns(); true);
```

- Counting the number of expressions that do/don't satisfy the predicate
    1. A majority of the simpler use cases in the \num_of quantifier fall into this category. We give multiple examples below (here) of this kind of use, so we will exclude including them here to avoid redundancy.
- Ensuring objects are instantiated
    1. Suppose, for example, we wish to populate a list of arbitrary data structures the length of which is known to us (i.e. 30 linked list nodes). If we specify a \num_of quantifier as follows:

$$(\backslash num\_of\ int\ i;\ 0 <= i\ \&\&\ i <\ DsList.length();\ isInstantiated(DsList[i]) \tag{1}$$

    We can verify that every object/data-stucture in this list is indeed instantiated.

In the following example we create a function that is intended to count the number of even numbers in an arbitrary integer array. We then go on to specify said function in openJML by saying 'ensure' the result is equal to the number of elements that satisfy $i \ \% \ 2 == 0$ within the range from zero, and the array length.

```
//@ ensures result == (\num_of int i; 0 < i && i <= array.length; (i % 2 == 0));
int counteven(int array[]){
    int j = 0;
    for(int i; i < array.length; i++){
           if(array[i] % 2 == 0) j++;
    }
    return j;
}
```

The \num_of quantifier in this instance maps the predicate over the given range (0 - ArrayLength) and then aggregates the result using the \sum quantifier specified to sum 1L for each satisfying value.

In the following example we create a slightly more complicated function that is intended to count the number of prime numbers before a given element n. In the interest of space, I've excluded the function bodies and only included the function headers. We then specify said function by ensuring the result is equal to the number of elements that satisfy (or return true for) isPrime(n) within the range of zero to n-1.

```
    (defun isPrime (n) "Is n prime? (Boolean)" body)
```

```
    //@ ensures \result == (\num_of int i; i < n; isPrime(i));
    public int countPrimes(int n) {body}
```

In the following and final example, we have a simple function that returns the number of values that are greater than the mean of an array. We specify this function by iterating through each element in the array and grabbing the number of elements that are above the mean. This value is then compared to the return value of the function stored in \result.

```
//@ \result == (\num_of int i; 0 < i && i <= array.length; (array[i] >
    mean(array)) );
int aboveMean(int array[]){
    ArrayList<Integer> retVal = new ArrayList<Integer>();
    for(int i=0; i<arr.length; i++){
        total = total + array[i];
    }

    double average = total / array.length;
    // Remove values less than mean
    retVal = Arrays.stream(array).filter(x -> x < average).toArray()
    return retVal.length;
}
```

6.3.4 Edge Cases

The \num_of quantifier is lucky in that it's potential to exhibit strange behavior is limited. By default, the body type is Java's long int, meaning errors involving overflow are non-practical. This is further reinforced by the fact that the internal sum only increases by one when the predicate is satisfied; in other words, $(9,223,372,036,854,775,807 \times 2 + 1)$ elements would need to be iterated to find issue with overflow.

Although impractical \num_of does in technicality suffer from all of the same edge cases that the \sum quantifier would suffer albeit to a lesser extent.

There is however something to be said regarding the range and the predicate, both of which must be satisfiable, to satisfy the \sum quantifier. Admittedly this is less of an edge case and more of a constraint imposed by OpenJML.

6.3.5 Requirements

There are few requirements to prove the correct implementation of the \num_of quantifier. First and most obviously, it should verify the number of elements that satisfy an arbitrary satisfiable predicate. Secondly, it must be identical in function to:

```
(\num_of T x; R(x); P(x)) == (\sum T x; R(x) && P(x); 1L);
```

This can be accomplished in the simplest sense, by making the \num_of quantifier parse down into its equivalent sum statement.

This is not to say that the underlying sum implementation will be trivial. A large portion of the work for this quantifier (and derivatively \sum) lies in the fact that SMT-Solvers currently do not have a neat way to express sums at all. The major challenge in implementation for our team is discovering this method. Our preliminary research shows that Z3 in particular does support, in the latest version, a form of recursion. Our goal is to make use of this recursion to implement the sum quantifier.

6.4 \product

6.4.1 Introduction

The \product Quantifier will multiply together elements at iterations that satisfy the range expression.

```
(\product int i; 0 < i && i < 5; i);
```

This is an example of a simple product quantifier comprehension expression. The product quantifier acts very similarly to the sum quantifier. The quantified variable, i, is in this case an integer. The range is from zero to five exclusive. The quantified expression is simply i so there are no intermediary calculations. This would be evaluated as follows.

1. When $i = 0$ then $0 < 0$ is false, and $0 < 5$ is true so the current total is 0
2. When $i = 1$ then $0 < 1$ is true, and $1 < 5$ is true so the current total is 1

3. When $i = 2$ then $0 < 2$ is true, and $2 < 5$ is true so the current total is 2×1
4. When $i = 3$ then $0 < 3$ is true, and $3 < 5$ is true so the current total is $3 \times 2 \times 1$
5. When $i = 4$ then $0 < 4$ is true, and $1 < 4$ is true so the current total is $4 \times 3 \times 2 \times 1$
6. When $i = 5$ then $0 < 5$ is true, and $5 < 5$ is false so the current total is $4 \times 3 \times 2 \times 1$

With this, we can see that the expression evaluates to 24. Similarly to the sum expression, we might notice that we only demonstrated the computation on values from zero to five. Again, it is worth noting that this is not how the quantifier will be evaluated in static checking. the static checker will effectively test every single value. Differently, assuming some optimization on ranges within the run-time assertion checker, we might be able to reduce operations to be only those needed. In this case, only from $i = 1$ to $i = 4$. Mathematically, this quantifier could be represented as:

$$\prod_{x \in R} P(x)$$

This mathematical grounding makes \product useful for many computation tasks. Some functions can be approximated with an infinite product such as:

$$\sin(x) = x \prod_{n=1}^{\infty} \left(1 - \frac{x^2}{\pi^2 n^2}\right)$$

6.4.2 Example

With some basic knowledge of how this product might be computed, we will break down a very simple test case. The product expression is less common than the broadly used sum. While many product expressions will have a complex body, in this case, we will keep it simple. This will make it a good baseline test case during implementation.

```
//@ requires 0 <= fac <50;
//@ ensures \result == (\product int i; 0<i<=fac; i);
public static int factorial(int fac) {
    int factorial = 1;

    //@ maintaining factorial == (\product short j; 0<j<=i ; j);
```

```
    for(int i=n; i>0; i--) {
        factorial*=i;
    }
    return factorial;
}
```

Within this example, we compute the factorial given an argument. To compute this and verify, we use the product quantifier two times. In order to keep the product from being too large, we require that the factorial we compute be less than fifty. We do allow factorial zero to be computed. As you might notice the function will return one if zero is the input. Given that a product quantification with an always false range returns one, this would not cause any issue. We also ensure that the resulting return value of the function is equal to the product of all of the numbers between the input and one. Then within the method, we use a loop variant to confirm that at each iteration we are computing the product correctly.

We would expect this test case to verify. Thus, for each possible and allowed input, we will return valid output. To have both a positive and negative test case, let us create one simple case that should not verify.

```
//@ requires 0 <= fac < 50;
//@ ensures \result == (\product int i; 0<i<=fac; i);
public static int factorial(int fac) {
    int factorial = 0;

    //@ maintaining factorial == (\product short j; 0<j<=i ; j);
    for(int i=n; i>0; i--) {
        factorial*=i;
    }
    return factorial;
}
```

By simply changing the initialization of factorial from one to zero we know that this case should not verify. Given that any input will return zero, the method is not returning the correct value.

6.4.3 Example Use Cases

Providing use cases for this quantifier will help the project in two ways. First it will give a larger set of tests for us to be able to develop the quantifiers with. It will also provide an example for future users on how they might go about using this quantifier in their code. It is important to have some examples for users to refer to so that they can better understand the syntax and use case.

In the field of computer science, we compute boolean arithmetic often. As you may know, the and calculation can be computed, and is often represented as a product. If we restrict the users to only enter the bits zero and one within an array, we can compute an and operation over the values within the array.

```
//@ requires (\forall int i; 0 <= i <args.length; args[i]==0 || args[i]==1);
//@ ensures \result == (\product int i; 0 <= i < args.length; args[i]);
public static int and(int args) {
    byte and = 1;

    //@ maintaining (\product int j; 0 <= j < args.length; args[j]);
    for(int i=0; i < args.length; i++) {
        and *= i;
    }
    return and;
}
```

As in languages such as c, one will represent true, while zero represents false. Given that anything multiplied by zero is zero if any of the values in the array are zero, the result is zero. This means that if all elements do not represent true within the array, we will return false.

To verify this, we first require that all elements are either zero or one within the array. We did this using the for all quantifier. Next, we ensure that the result is the product of the elements within the input array. We do the same check within the function at each iteration.

Another common use of multiplication is to compute a power. Given that the computation of a power is the multiplication of the same element into a product n times where n is the power, we can use a product to represent this function.

```
//@ ensures \result == (\product int i; 0<i<power; base);
public static int calcPower(int base, int power) {
    int res = 1;

    //@ maintaining (\product int j; 0<j<=power; base);
    for(int i=0; i<args.length; i++) {
        res*=base;
    }
    return and;
}
```

To compute the power we take the input base and power from the user. Our goal is to compute a product where the base is multiplied into itself the amount of times of the power. To do this, we created a product quantifier that runs from one to the power value. At each iteration, our quantified expression is simply base. Thus, we are multiplying the base into the product as we wanted.

6.4.4 Edge Cases

In order to verify that the product quantifier is properly handling all cases, we need to test the bounds of the quantifier.

- NaN is in an array we are calculating over
- Overflowing the result
- Range expression is always True
- Range expression is always False
- Multiply over an array of user defined objects on a specific field
- Multiply order of floats that might not be commutative in the IEEE-754 standard

For the Run-time Assertion Checker a few key cases to test:

- Range expression produces a large set of iterations that is unpractical to compute at run-time.
- The range expression produces multiple sets that do not intersect

6.4.5 Requirements

The \product quantifier must maintain a number of requirements imposed by the JML specification language. First and most obviously, it should verify the product of a set of elements that satisfy an arbitrary satisfiable predicate. That is to say it should properly take the product over a given array-like structure.

Similarly to \sum, this proves difficult only because SMT-Solvers do not have the innate capability to reason about sums and products. For this quantifier to work properly, our group must implement a pipeline that allows products to be parsed into SMT and subsequently understood by SMT solvers. The same pipeline created for \sum can be reused here with a slight change to arithmetic.

6.5 \min

6.5.1 Introduction

The \min quantifier validates the minimum value within a specified range that satisfy a given Boolean expression. \min can include different data-types such as: integers, objects and floats.

\min is a quantifier that has the general form

$$(\backslash min\ t_1\ i,\ t_2\ j,\ \ldots\ t_n\ r;\ a;\ b)$$

where

- t_i is the type $t_(i+1)$.
- a is a Boolean predicate
- b is a numeric type which can be seen as a function f where the range of f is over a numeric type

\min can be seen as a function g defined the following way, g takes as input the set of numeric types d that satisfy:

$$f(c_1, c_2, \ldots, c_n) = d$$

for some c_1,c_2,...,c_n of types t_1,t_2, ...,t_n respectively and where

$$h(c_1, c_2, \ldots, c_n) = True$$

and returns the minimum.
\min can be used for codes that are computing the minimum value of a range of values.

6.5.2 Example

- (\min $int\ i,\ intj; 0 <= i\ \&\&\ i < 10\ \&\&\ 0 <= j\ \&\&\ j < 10;\ a[i][j]$)
- (\min $int\ i,\ intj; 0 <= i\ \&\&\ i < 10\ \&\&\ i <= j\ \&\&\ j < 2 * i;\ a[i][j]$)
- (\min $int\ i;\ 0 <= i\ \&\&\ i < 5;$ (\min $int\ j;\ 0 <= j\ \&\&\ j < i;\ map.get(j)$)

6.5.3 Example Use Cases

- Maximum value in a collection - find the max of a list of values, such as an array or linked-list.
- Selection sort - on every iteration of the sort, the selected item must be the minimum (or maximum depending on the order). Although there are better sorting algorithms, this method can be a useful teaching tool and can be modified for specialized situations.
- Least Common Multiple - of all divisors for two values, the returned value should be the minimum.
- Minimum profit calculations - given prices over n days, compute the minimum profit that can be gained by buying and selling at most k times.

```
public class Min {
    //@ requires 0 < a.length < Integer.MAX_VALUE && a != null ;
    //@ ensures (\min int j; 0 <= j && j < a.length; a[j]) == \result;
    public int arrayMin(int[] a)
    {
        int min;
        min = a[0];

        //@ loop_invariant 0 <= i <= a.length;
        //@ loop_invariant (\min int j; 0 <= j && j < i; a[j]) == min;
```

```
        for(int i = 0; i < a.length; i++)
        {
            if(a[i] < min)
            {
                min = a[i];
            }
        }

        return min;
    }
}
```

This test case tests whether the function arrayMin returns the minimum in an array using the \min quantifier. The first OpenJML comment is a precondition. This means it requires precondition to be satisfied to evaluate the postcondition. The length of the array has to be greater than 0 and less than Integer.MAX_VALUE and that array a cannot be null. So the requires comment says to evaluate the postcondition any time the requires statement evaluates to true. The ensures comment is the postcondition and it states that the result of the function is equal to the minimum of the array, the solver will evaluate whether the precondition implies the postcondition. Inside the function there are the loop invariants. This evaluate whether the statement that follows it is true before and after every execution of one loop of the for loop. The first invariant asserts that i is always equal or greater than 0 and equal or less than the length of the array. The second asserts the current minimum in the loop is equal to min.

```
import java.util.ArrayList;

public class MinList
{

    //@ requires 0 < a.size() < Integer.MAX_VALUE && a != null ;
    //@ ensures (\min int j; 0 <= j && j < a.size(); a.get(j)) == \result;
    public int listMin(ArrayList<Integer> a)
    {
        int min;
        min = a.get(0);

        //@ loop_invariant 0 <= i <= a.size();
```

```
        //@ loop_invariant (\min int j; 0 <= j && j < i; a.get(j)) == min;
        for(int i = 0; i < a.size(); i++)
        {
            if(a.get(i) < min)
            {
                min = a.get(i);
            }
        }

        return min;
    }
}
```

This test case tests whether the function ListMin returns the minimum in an arrayList using the \min quantifier. It is very similar to the previous test case but shows that one can evaluate the behavior of different data structures using OpenJML. As you can see in the OpenJML comments [] was substituted by the function get() which is used to access members of the arrayList and the field length was substituted by the function size() which returns the number of elements in the arrayList.

6.5.4 Edge Cases

In order to verify that the \min quantifier is properly handling all cases, we need to test the bounds of the quantifier.

- NaN is in an array we are calculating over
- Range expression is always True
- Range expression is always False
- Achieve minimum of user defined objects on a specific field

For the Run-time Assertion Checker a few key cases to test:

- Range expression produces a large set of iterations that is unpractical to compute at run-time.
- The range expression produces sets that do not intersect

6.5.5 Requirements

For \min the types of the variables do not need to be numeric types they can be objects or Boolean. For example, you could quantify over arrays or lists of a created object. b does need to be a numeric type and a does need to be a Boolean type. There can hypothetically be as many variables as one wants but of course one needs to remember there is a memory limit. a can also be always false in which case a would be unsatisfiable. If a is unsatisfiable then the value of \min will depend on the type of b. If b is an integer then we say \min returns undefined, if b is a float then it will return Float.NEGATIVE_INFINITY. b can also be a constant in which case if a is satisfiable \min would just return the value that b always maps to. \min should also be able to handle different data structures being part of the body, for example arrays, hash-maps, hash-sets, Lists and created objects that store numeric types. \min like any other quantifier should also be able to handle nested quantifiers.

\min needs to be able to return the minimum of numeric types and it can be used as a specification for programs that attempt to do the following:

- Returning the minimum of an array
- Define a Min Priority queue
- Find the minimum in a tree structure
- Return the minimum is a set
- Search problems where one is looking for the minimum cost path
- Problems to find the minimum cost option, for example for companies
- Find the least recently used address in a virtual cache
- Find the Least common multiplier

We will need to create various test cases to see whether the specifications are working as intended, the test cases will consist of java code together with specifications. We will create test cases that are supposed to be valid and test cases that are not supposed to be valid.

Test cases to consider for \min:

- Collections that have positive and negative numbers

- Collections with floats and doubles
- Collections with different types of data structures (example hash sets, arrays, lists, trees)
- Collections containing INTEGER.MIN_VALUE and FLOAT.NEGATIVE_INFINITY
- Collections such that in $(\min tx; a; b)$ b is a function of one or more elements example b could be $i + j$
- Collections with even and odd numbers
- Collections that contain 0
- (\min $t\ x;\ a;\ b$) where a is always false
- (\min $t\ x;\ a;\ b$) where x is not a free variable in a
- (\min $t\ x;\ a;\ b$) where the relational operators in a are == and < and && and ||
- Using NaN
- Convergence
- Nested quantifiers (combining \min and \max or \min and \product or \min and \sum or \min and \num_of or \min and \min)
- Nested quantifiers where \forall and \exist quantifiers are put in a.

6.6 \max

6.6.1 Introduction

As the name suggests, the \max quantifier returns the maximum of the values of the given expression where the variables satisfy a condition.

The general form is as follows:

```
(\max T x; R(x); P(x));
```

The function $R(x)$ returns a boolean indicating if variables x satisfies some condition. The function $P(x)$ is the expression for which the maximum should be found.

```
(\max int i; 0 <= i && i < 5; i) == max(0, 1, 2, 3, 4) == 4;
```

$P(x)$ will be evaluated with 'mathematical arithmetics' and coerced to the correct type using the math mode in effect. Valid types are any of the Java numeric types such as short, int, long, float, and double.

If $R(x)$ is always false (also referred to as not satisfiable), the \max quantifier should either return negative infinity or throw an error.

$$\begin{cases} -\infty & \text{type}(P(x)) \in \{float, double\} \\ undef & \text{type}(P(x)) \in \{byte, int, long, bigint\} \end{cases}$$

If $R(x)$ is always true (also referred to as always satisfiable), the \max quantifier should return the maximum value as allowed by the type of $P(x)$. For floating point values that is $-\infty$ and for integer values it is the MAX_VALUE. However, during runtime assertion checking this should warn the user because iterating over every possible value of x is both infeasible and likely incorrect. A 'well-defined' range should always be preferred.

The last important edge case is floating point values. The IEEE 754 floating-point standard includes a Not-a-Number value (Float.NaN or Double.NaN). All comparisons with NaN return false in Java. Specifically, the following holds:

$$NaN < 1.0 \Rightarrow false$$
$$NaN > 1.0 \Rightarrow false$$

which obviously can cause problems when trying to find a maximum value. If $P(x)$ can produce NaN for any x that satisfies $R(x)$, the result of the quantifier should be NaN. The Java Math class follows this rule for the Math.max method [12]. If this is not desirable, the user can exclude NaN in $R(x)$ with such a condition (!Double.isNaN($arr[x]$)).

6.6.2 Example

Although the \max quantifier does not exist in OpenJML, a work around can be used to prove a value is maximal. First, one must show the value is greater than all other values in

the range. Second, one must show that the value is in the range.

```
//@ ensures (\forall int i; 0 <= i && i < array.length; array[i] <= \result);
//@ ensures (\exists int i; 0 <= i && i < array.length; array[i] == \result);
public int getMax(int[] array) {
    int bestMax = array[0];

    //@ loop_invariant 0 <= i <= array.length;
    //@ loop_invariant (\forall int j; 0 <= j < i; array[j] <= currentMax);
    //@ loop_invariant (\exists int j; 0 <= j < i; array[j] == currentMax);
    //@ loop_decreases array.length - i;
    for (int i = 1; i < array.length; i++)
        if (bestMax < array[i])
            bestMax = array[i];
    return bestMax;
}
```

This method is clunky and obscures the true function specification. The addition of \max will allow specifications such as:

```
//@ ensures (\max int i; 0 <= i && i < array.length; array[i]) == \result;
public int getMax(int[] array) {
    int bestMax = array[0];

    //@ loop_invariant 0 <= i <= array.length;
    //@ loop_invariant (\max int j; 0 <= j && j < i; array[j]) == currentMax;
    //@ loop_decreases array.length - i;
    for (int i = 1; i < array.length; i++)
        if (bestMax < array[i])
            bestMax = array[i];
    return bestMax;
}
```

This equivalence can be exploited during static checking. Proving the value is greater than or equal to all values in the collection and the value is in the collection demonstrates the value is maximal.

6.6.3 Example Use Cases

- Maximum value in a collection - find the max of a list of values, such as an array or linked-list.
- Selection sort - on every iteration of the sort, the selected item must be the maximum (or minimum depending on the order). Although there are better sorting algorithms, this method can be a useful teaching tool and can be modified for specialized situations.
- Priority queue - ensure the pulled item always has the highest priority. This data structure is used in many algorithms such as Dijkstra's algorithm.
- Maximum flow - ensure the returned route has a much flow as possible from the source to the sink.
- Greatest common divisor - of all divisors for two values, the returned value should be the maximum.
- Maximum profit calculations - given prices over n days, compute the maximum profit that can be gained by buying and selling at most k times.

The introduction of streams in Java 8 and the continued optimization efforts for them indicates that operating over ranges of data is important.

Reduction operations are one important feature of streams that allows one to reduce data to a single value using some standard operation.

```
// Using conventional for loops
int max = numbers[0];
for (int i = 0; i < numbers.length; i++) {
    if (numbers[i] > max) max = numbers[i];
}

// Using Java8 Streams
int max = numbers.stream().max();
```

Using streams can result in faster and clearer code and as such the program verification should be similar. Reducing the amount of logic needed to achieve the programs functional intent is important for bug free code. Using streams helps mitigate off by one errors on arrays and can allow for easy parallel processing. Additionally, \max will allow easier verification

of the Java standard library which is necessary for full confidence in program correctness. Keeping up with the latest long term support versions of Java is critical for OpenJML to maintain relevance.

6.6.4 Edge Cases

Normal cases

The first set of test cases will simulate the most common use cases for the \max quantifier. Functions will be created that involve the following cases. For each case there will be a test that should and should not verify

- P returns an int
- P returns a long
- P returns a short
- P returns a float
- P returns a double
- P can return positive and negative values
- P uses arithmetic operators
- P indexes into an array of type T
- P gets a value from a List at an index
- P gets a value from a List at a key
- P uses multiple variables
- P is another Java function
- P involves overflowing or under flowing an Integer value
- R is the conjunction of multiple ranges
- R uses multiple variables
- R is another Java function

- \max is nested inside another quantifier
- Another quantifier is nested inside \max

Edge cases

The next set of test cases will simulate abnormal edge cases.

- $P(x) = NaN$ for some x
- $P(x)$ throws an exception for some x, such as $P(x) = 100/x$
- R is always True
- R is always False
- $R(x)$ throws an exception for some x, such as $R(x) = 90\%x == 50\%x$
- x has the type of a User defined Object
- x has the type of a Java defined Object such as String

For the Runtime Assertion Checker a few key cases to test:

- $R(x)$ is a sufficiently large set such that it would not be practical to iterate over the entire range
- $R(x)$ is a disjoint set of multiple ranges
- x is non-numeric

Particularly, when translating JML annotations to pre and post assertions, the resulting functions should be viably executable. If R is always true, iterating over every possible x may take too long. The user should be warned of this behavior.

In addition, R can have many forms that are equivalent. The assertion checker must be able to parse all ways of arranging the condition.

- $0 < i$ && $i < 10$
- $0 < i < 10$
- $!(i < 1 \mid\mid i > 9)$
- $1 <= i$ && $9 >= i$

6.6.5 Requirements

For \max the types of the variables do not need to be numeric types they can be objects or Boolean. For example, you could quantify over arrays or lists of a created object. b does need to be a numeric type and a does need to be a Boolean type. There can hypothetically be as many variables as one wants but of course one needs to remember there is a memory limit. a can also be always false in which case a would be unsatisfiable. If a is unsatisfiable then the value of \max will depend on the type of b. If b is an integer then we say \max returns undefined, if b is a float then it will return Float.NEGATIVE_INFINITY. b can also be a constant in which case if a is satisfiable \max would just return the value that b always maps to. \max should also be able to handle different data structures being part of the body, for example arrays, hash-maps, hash-sets, Lists and created objects that store numeric types. \max like any other quantifier should also be able to handle nested quantifiers.

7 Implementation

7.1 Test Cases in OpenJML

There are two kinds of test cases in OpenJML both of which use the JUnit testing framework. The 'JmlTestCase' abstract class provides the basic setup and tear-down code. The 'Main' OpenJML class is constructed. Child classes can add options to the instance. The 'ESCBase' includes helper functions for running esc on files and matching error messages to expected messages. In addition, the tests can optionally be run on multiple SMT-solvers via JUnit parameters. This will be important to ensure the SMT translations are not solver specific.

Any classes in the 'org.jmlspecs.openjmltest' namespace may include multiple tests as denoted by a new function name and an '@Test' annotation. Tests in the 'test' folder should test a single specific feature and provide comments explaining what is being tested. Runtime assertion checker tests should include two expected files: one for compile-time output and one for runtime output.

In general, the tests used for extended static checking can be reused for runtime assertion checking; however, there are different edge cases for both. When writing test cases, there will always be separate folders for both tools as their outputs and use cases are different. This will help ensure full coverage for both tools even if they diverge in support.

Run the 'AllTests' class to run the JUnit testing suite on all test cases in the 'test cases' folder. This will produce a report of the success of every test.

7.1.1 Testing a Single Java Class

These tests are in 'OpenJMLTest/test'. Each folder contains a single Java Class testing some functionality of OpenJML. The folder also contains 'expected' text files of the output. The class should be a contained unit that can be run on OpenJML. These tests can also serve as examples of how to use an OpenJML feature.

'escAdd/Add.java' is one such example:

```
// This example had a problem with crashing, because of the lack of helper
// on the functions used in the invariant.
public class Add
{
```

```
  //@ public invariant x() + y() > 0;

  private /*@ spec_public */ int my_x;
  private /*@ spec_public */ int my_y;

  //@ requires the_x + the_y > 0;
  //@ ensures x() == the_x && y() == the_y;
  public Add(final int the_x, final int the_y)
  {
    my_x = the_x;
    my_y = the_y;
  }

  //@ ensures \result == my_x;
  public /*@ pure @*/ int x() { return my_x; }
  //@ ensures \result == my_y;
  public /*@ pure @*/ int y() { return my_y; }

  //@ ensures \result == x() + y() + the_operand;
  public /*@ pure @*/ int sum(final int the_operand)
  {
    return my_x + my_y + the_operand;
  }
}
```

To run these tests automatically, one must add a reference to a file to the runner code in 'org.jmlspecs.openjmltest.testcases'.

```
@Test
public void testAdd() {
    expectedExit = 1;
    helpTF("escAdd");
}
```

The 'help' collection of functions allows specification of command-line arguments (such as verbosity, esc, and rac). 'expectedExit' should be set to 1 if the program should not verify and 0 if it should. In the case above, 'helpTF' expands to 'helpTCF' which checks if the value returned by 'Main.execute' equals 'expectedExit'.

Once added, the JUnit testing framework will automatically test the class and check if the output is as expected.

7.1.2 Class as a String

Another, more painful, way of writing tests is to hard code a Java Class as a String in a JUnit test.

```
@Test
public void testX() {
    main.addOptions("-method=m1");
    helpTCX("tt.TestJava","package tt; \n"
          +"public class TestJava { \n"
          +"  public void m1(Object o) {\n"
          +"    //@ ghost array<Object> a; \n"
          +"    //@ set a[3] = o;\n"
          +"  }\n"
          +"}"
          );
}
```

This type should be reserved for very concise tests as it is generally harder to read, and requires the rest of the testing apparatus to function. These types of tests are useful when additional code may be required to run a test. When possible, the underlying class should be extracted into a separate file.

7.2 Python Z3 Package

In order to understand this process, we will use Python to create SMT-LIB code that will compute the quantifier expressions. By writing the first few iterations with a familiar syntax like Python, using the Z3 package, we will be able to better grasp the problem. With a working quantifier within Python with the Z3 package, we will convert the JML test cases to SMT-LIB. We might use Python as an intermediary to make this process easier.

Once we have translated everything into SMT-LIB we will have sets of logical operations that are equivalent to the individual quantifiers. With the pre-constructed nodes in the

OpenJML library, we should be able to construct a new node for our quantifiers. These nodes contain the logic to translate to the other forms within the process and eventually SMT-LIB. These will be groups of nodes that compute a specific logical operation or set of operations given an input. Then, with a new node for each quantifier, we just need that node to be called given when the token is found for the quantifier. With the parsing and tokenization for the quantifiers already being implemented, we strictly need to focus on the formal logic and the construction of the nodes.

7.3 Extended Static Checker

Implementing the new quantifiers in OpenJML's extended static checker was done in multiple phases. This began by writing three individual test cases for each quantifier. From there, our group expanded the number of test cases to improve coverage and cover possible edge cases as expressed above. With the test cases fully developed, the project now had a firm foundation from which we could continue with the manual SMT translations.

The key to this portion of implementation was being able to convert a quantifier into simpler functions and expressions that the solvers can interpret. Important factors contributing to this implementation were addressed by Leino and Monahan. Namely, for all three of the quantifiers it would not be possible to directly include arithmetic symbols $(*, +)$ [5]. For our project to successfully run on Z3, we had to work around this constraint in developing our hand translations.

The implementation of the quantifiers in the ESC began with the ground-work discussed in [5]. The quantifiers were declared as functions within SMT-LIB and then assertions that the function must satisfy a set of axioms were made. At first, we started with simple axioms like induction from above axiom and a base case axiom. This implementation was passable but required pre-defined ranges of values as input which was less than idea. The first natural point of progression was implementing the rest of the axioms discussed in the Leino and Monahan paper. Even with the inclusion of all axioms, the solution would not halt in certain situations. Namely, we could achieve a halt on SAT problems only or vice versa.

After many attempts to accommodate the Leino axioms into our implementation, our group decided to try a different approach by using the

```
(define-fun-rec f ((x_1 t_1 ... x_n t_n)) y t)
```

syntax. This particular syntax is only in newer versions of z3 and allowed us to define a function recursively. Using this implementation we were able to forgo defining axioms as they are included in the definition of the function. This approach halted when appropriate for all test-cases and thus our group continued development using recursion.

The following are rules that define how the translations from OpenJML to SMT-LIB Work. Sum:

```
[[|(\sum T x; Range; Body)|]] =>
    (define-fun Range_N (x T) Bool [[Range]])
    (define-fun Body_N (x T) K [[Body]])

    (define-fun-rec sum_N
        ((lo K) (hi K)) K
        (ite (< hi lo)
            0
            (+
                (sum_N lo (- hi 1))
                (ite (Range_N hi)
                    (Body_N hi)
                    0
                )
            )
        )
    )
```

Product:

```
[[|(\product T x; Range; Body)|]] =>
    (define-fun Range_N (x T) Bool [[Range]])
    (define-fun Body_N (x T) K [[Body]])

    (define-fun-rec product_N
        ((lo K) (hi K)) K
        (ite (< hi lo)
            1
            (*
                (sum_N lo (- hi 1))
                (ite (Range_N hi)
```

```
                (Body_N hi)
                1
            )
        )
    )
)
```

num of:

```
[[|(\num_of T x; Range; Body)|]]
  =>
    [[|(\sum T x; Range && Body; 1)|]]
```

As it can be seen, these translations only work for quantification over one variable. For quantification over multiple variables we developed a translation that uses the concept of currying, so model a function of multiple arguments as functions of one argument which return other functions of one argument. The formal translation is the following:

```
[[|(Q T x1, x2, ..., xn; Range; Body)|]] =>
  [[(Q T x1; ; [[(Q T x2, ..., xn; Range; Body)]])]]

[[|(Q T x; ; Body)|]] => [[|(Q T x; true; Body)|]]
```

We used built-in functions in the OpenJML repository to implement the sum translation. The code allows for the OpenJML specifications to be translated into SMT-LIB for the solver to analyze. We wrote the code in the SMTTranslator class of OpenJML and the OpenJDKModule for OpenJML. The code in the SMTTranslator class parses and translates the specifications to SMT-LIB syntax in the same way the translations specify above. The code in the OpenJDKModule allows for the declaration and parsing of recursive functions which is syntax from the newer versions of the SMT-LIB standard.

By implementing the translation in OpenJML we can verify the correctness and incorrectness of certain Java programs that use the sum quantifier. A simple example of a program that we can verify is incorrect is a program that adds the numbers from 0 to n by specifying the sum should be from 1 to n+1.

7.4 Run-time Assertion Checker

7.4.1 Run-time Assertion Checking Quantifier Comprehensions

By implementing run time assertion checking for quantifier comprehensions we hoped to provide a quick verification technique that will reduce verbosity of OpenJML. Given that run-time assertion checking is focused on being easier and quicker than static checking, having the ability to check large chunks of code in one line is a significant benefit. By reducing the time and effort to write assertions, we can increase the benefit per unit development time.

As we have learned, OpenJML adds in functions to the Java code and compiles them into the byte code in order to check specific assertions at run-time. With this, we know that in order to implement run-time assertion checking, we will need to create a logic to translate the OpenJML assertions into Java code.

Run-time assertion checking quantifier comprehensions requires a way to iterate over a range and execute an expression at each iteration. There are many ways to write functions that iterate over ranges within Java. To name a few, there are for loops, while loops, recursion, and iterators. It is important to note that some techniques of iteration might be more efficient for some situations than others. For example, a recursive function takes more memory to run, and could eventually cause you to run out of heap space if it operates on enough iterations. With this in mind, we want to optimize our assertion checking to be both most efficient, but also make sure that the ranges are well defined.

With the knowledge that we will have to build the logic to create Java methods from JML annotations for the quantifiers, we need to understand what logic we can extract from the quantifiers to build our methods. Each of the five quantifiers have three main parts, the quantified variable, the range, and the quantified expression.

```
//@ (\num_of int i; 0 <= i < 7; arr[i]<10)
int[] arr = {18, 19, 1, 29, 9, 21, 4}
```

With the example quantifier comprehension above and the given array, there are multiple ways we could write a java function to compute the expression.

```
private static int forLoop(int[] arr) {
    int count = 0;
    for(int i=0; i<7; i++) {
```

```
        if(arr[i] < 10) {
            count++;
        }
    }
    return count;
}

private static int whileLoop(int[] arr) {
    int count = 0;
    int i = 0;
    while(i<7) {
        if(arr[i] < 10) {
            count++;
        }
        i++;
    }
    return count;
}

private static int recur(int[] arr, i) {
    if(arr[i] < 10) {
        return 1 + recur(arr[i+1], i+1);
    }
    return recur(arr[i+1], i+1);
}

private static int forEachLoop(int[] arr) {
    int count = 0;
    for(int elem: arr) {
        if(elem < 10) {
            count++;
        }
    }
    return count;
}
```

Each of the above functions does the exact same thing, just in different ways. However, some of the methods do not fully incorporate all three parts of the quantifier comprehension. For example, the method recur and forEachLoop are not incorporating the range expression.

Yes, both could have an if statement or some other check to verify that the index is within the range, but they do not in this case. This demonstrates that even if we can create a method that works for one comprehension, we need to create a translation that will equate to the OpenJML expression directly.

Knowing that our goal is to build something that incorporates all three of the parts of the compression, we want to take this concept and make it as generic as possible. What is the shell of the function that we can use to construct any possible quantifier comprehension of a given type from the annotation. Given that each quantifier is unique, they should require slightly different implementations with overlap in some cases.

For the \min and \max quantifiers, we will need to keep track of the value that is currently the minimum or maximum. We will then need to loop over the range. At each iteration, we will compare the current maximum or minimum to the quantified expression at the given index.

With \sum, \num_of, and \product we will need to store and calculate the value over the range. We should be able to accomplish this for sum and product by inserting the quantified expression within the loop body, and iterating over the range. With number of, we can abstract sum to represent number of, where the quantified expression is simply one. However, the range will be slightly more complicated because we will need to append the expression for number of onto the range of the sum we abstract to.

To represent this in code, we will create function stubs and add on the requirements based off the quantifier, and its inputs.

7.4.2 Quantifier Comprehensions Ranges

The run-time assertion checker needs to compute ranges effectively. Given the complex syntax that OpenJML supports, as well as user creativity, understanding and effectively translating ranges into compiled Java code can be complex.

It is worth reiterating that the range expressions are simply booleans that determine whether a number is to be considered within the expression. This means that the expression does not have to follow the x is greater than zero and less than five notation. Given the complexity of these expressions, it would be best to find the bounds to not test every single value of a given data type. In run-time assertion checking it is important to not cripple the speed of the underlying program. Given that this is supposed to be the quick and easy alternative to

static checking, it must run more efficiently.

Run-time assertion checking relies on run-time values within the program. This is different than static checking which will verify over all possible inputs. Since we know the input that we will be testing, we have the opportunity to simplify the concept of a range. At the same time, it is worth noting that there is no defined iteration order within the JML syntax.

Where a Java for loop gives you the opportunity to call an expression which iterates to the next instance in the loop. JML assumes that the loop will evaluate each value over the range, with no specified ordering unless defined by the range. With this, and the complexity that can be put into a boolean expression, we must be able to evaluate these expressions efficiently to iterate over the least possible number of inputs.

Lets look at example of a function using a more complex range.

```
//@ ensures \result == (\sum int i; i%2==0 && 0<=i<=n; i);
public static int SumEven(int n) {
    int sum = 0;

    //@ maintaining sum == (\sum int j; j%2==0 && 0<=j<=n; j);
    for(int i=0; i<=n; i++) {
        if(i%2==0) {
            sum+=i;
        }
    }
    return sum;
}
```

In the above example to write a Java function equivalent to the range expression we had to add an if statement. Importantly, this draws a major difference in the structure of the evaluation. Since a for loop has no range expression, instead it has a termination expression. This means rather than being a filter as the range expression is in JML, once the expression evaluates to false, the loop stops. This means that we cannot directly drop the range expression into the Java termination expression.

In order to deal with these complex ranges, we might need to modify how we evaluate the range expression. Importantly, we will still need to consider adding a upper and lower bound with which we can create our for loop. This will significantly reduce run time versus running on all available values within the data type and only selecting those which satisfy the if

statement.

We have now determined it would be best to construct an if statement, and use this to hold our range expression. At the same time, we know that we need to find the upper and lower bounds to the range to use within our for loop to reduce run time. This is where the complexity comes in. How can we quickly find the a value which satisfies the boolean where the next value greater or lower results in unsatisfied?

This problem could be addressed by iterating though the options one by one, and finding the bounds. This would result in faster and correct compiled java code, but the construction of this method might take significantly longer than if we did not add this feature. To solve issues where we need to find the inputs to a function that give a specific solution we use binary search. if we binary search for these min and max values we can significantly reduce compile time versus the linear search, and still get the great benefits of reducing the computation range at run time. There is a slight complication to this. There can be multiple local upper and lower bounds, and we just want the entire functions upper and lower bounds. Using a binary search could result in finding a local minimum or maximum and excluding values from the calculation range. We will first create the linear search implementation and then we can find ways to implement other more complex search techniques such as a modified binary search where we can reduce compilation times.

Once we complete this portion of the run-time assertion checking implementation, we can consider adding some new features such as iterators within the range expression as seen in the below example.

```
//@ ensures \result == (\sum Iterator<Integer> i; i.hasNext(); i);
public static int SumList(ArrayList<Integer> nums) {
    int sum = 0;

    //@ maintaining sum == (\sum Iterator<Integer> j; j.hasNext() ; j);
    for(int i: nums) {
        sum+=i;
    }

    return sum;
}
```

Most loops and iterating functions in enterprise Java use iterators. This could significantly

increase adoption within the community. In this portion of the implementation, we will need to consider how the quantified variable would differ from a traditional range, and how the range expression could be described with Java code.

8 Conclusions

8.1 Project Summary

Our team choose this project for many of the same collective reasons. We, as a unit, enjoy discrete math and problem-solving involving logic. While choosing from the available projects presented to us, the OpenJML project was the only one that fit within these interests. From the onset of our project to where it stands at the time of writing, our team has successfully cleared many hurdles and has achieved a satisfactory result. We aimed to complete the project beyond the expectations of our sponsor Dr. Leavens and we believe we have done so.

A major concern from the beginning of our project was creating an effective division of workflow, that allowed us to make steady progress towards completion. OpenJML involves a large number of topics that were foreign to us, therefore a major priority of ours was to fully understand the domain knowledge before continuing to actual development. It was clear then the first stage of our workflow must be research, without which we would not make any progress. Moving past this and as it stands now, we've further divided our workflow into two primary sections that roughly align with the two-semester long period in which we have to work on this project. The first of which is the earlier discussed research phase.

We performed some preliminary research so that we could accurately devise what subjects we needed to understand to move forward. Our research list consisted of:

- OpenJML Syntax
- OpenJML Class Structure
- SMT-LIB Syntax
- Software Verification
- Basic OpenJDK knowledge

- Basic working knowledge of compiler architecture (OpenJDK)
- OpenJML's verification Pipeline

If we could conquer these topics, to at least a basic level we were confident in our ability to apply this knowledge to actual development. From here we have our second stage, implementation. Although we did not begin the implementation phase until a few weeks into the second semester of Senior Design, our group set up a framework for success throughout the first semester allowing us to make excellent progress.

8.1.1 The Research Phase

The nature of the OpenJML project requires a research-intensive workflow. Our team as a unit possessed zero experience with SMT-Solvers, OpenJML, Software Verification and relied on our base-line knowledge in areas such as Discrete Mathematics to expand our knowledge base forward to cover the above-listed topics. The first semester, which is now coming to a close, consisted primarily of expanding this knowledge base under the guidance of Dr. Leavens and his associate Dr. Cok

At the beginning stage, our team attacked the problem from a division-of-knowledge perspective. The amount of required knowledge appeared to be too great to learn for one person in such a short time frame. As a result, we divided the knowledge base into sectors that were more digestible for each team member. This was effective in allowing us to cover as much of the knowledge base as possible, in the allotted time window.

8.1.2 The Implementation Phase

Furthermore, our group needed to understand more than just the underlying technology to move forward with implementation. Starting in early October, we shifted focus from a theoretical understanding to researching with implementation in mind. This meant focusing less on total domain coverage and rather understanding the internals of the OpenJML codebase in which we are to work on. After a lengthy discussion with Dr.Cok, we gained some useful insight into implementation that proved invaluable moving into semester two. Dr.Cok laid these out as follows:

1. Create a set of fairly simple tests that show use cases of what reasoning using \sum,

\product, \num_of should be supported.The examples should be small but realistic programs (e.g. loops to compute or count something)

2. Devise a method for these quantified expressions to be represented in SMT (following Monahan & Leino).Do some examples by hand, perhaps building from actual set output from OpenJML

3. Implement this design in the OpenJML Extended Static Checker

We used these steps given by Dr. Cok as a guideline for implementation and shifted research focus to best help us to achieve these goals. The steps as laid out by Dr. Cok threw a wrench in our earlier plans. Up until this point our group understood the task as implementing the 5 quantified expressions into the runtime assertion checker. After meeting with Dr. Cok, we discovered that they are already implemented for the most part in the RAC, and our major focus would be implementing them in the ESC. From here, we had to re-evaluate our project and our design paper and redesign them to fit this goal. Our primary focus from this point forward would be the ESC, and implementing the quantified expressions therein. Dr. Cok also expanded our reach goals slightly by giving us a bit of insight into what was not working in the RAC. Using these new guidelines, if extremely successful with our implementation, we could do further work into:

1. Expanding test examples of quantified expressions in the RAC

2. Evaluate bugs and points of error in the RAC

3. Repair said bugs

As our group moved into Senior Design 2, Spring semester of 2022, we began the actual implementation phase of our project. The time we spent researching allowed us a deal of freedom in terms of time to explore the methods by which we meant to implement the 5 major quantifiers in the Extended Static Checker.

Our group decided to let warnings of delaying development until Senior Design 2 go unheeded, and this decision has payed dividends in the rapid progress we were able to make on the problem. We were able to complete more and research far less. This is not to say that further research was not required. Research is an integral part of our project, and will be within any future Senior Design groups working on OpenJML. We planned our project in accordance with this fact.

Our first foray into implementation began with mock test-cases written in Java, annotated with the non-implemented quantifiers. Giving us something to work on while we bolstered our understanding of OpenJML through exposure. An example of one such test case is below. The full list of test cases developed can be found on our project's github page.

```
// expected : Unsat

public class sumArray
{
    // eval sum of array
    //@ requires 0 < arr.length < 10;
    //@ ensures \result == (\sum int i; 0 <= i && i < arr.length; arr[i]);
    public static int s3(int[] arr) {
        int total = 0;

        //@ maintaining 0 <= j <= arr.length;
        //@ maintaining total == (\sum int i; 0 <= i && i < j; arr[i]);
        //@ decreasing arr.length - j;
        for (int j = 0; j < arr.length; j++) {
            //@ assume Integer.MIN_VALUE <= total + arr[j] <= Integer.MAX_VALUE;
                // Just assume we never overflow
            total += arr[j];
        }

        return total;
    }

}
```

We established a number of consistent test-cases with the maximum amount of coverage possible. Coverage in this case, refers to the amount of possible use/edge cases our tests account for. Each quantifier had it's own mini-suite of tests created, ensuring we could move forward with our test-centric development plan.

From here, we focused on generating SMT conversions by hand so we could conceptualize the standard methodology of conversion that OpenJML would need to perform when implementing the quantifiers. Past this, generating the SMT-LIB by hand was also a good exercise in understanding.

Now that these test cases are developed, and their corresponding SMT translations can be proven using Z3 (or another popular SMT solver), we began to write the code required to get OpenJML to produce these SMT compliant statements. This process was fairly parallel where each group member will work on their quantifiers compilation process. There was minimal overlap in terms of actual code, but lots of theoretical overlap.

This began our foray into making a pipeline for converting the quantifiers into native SMT. The most obvious hurdle involved was achieving a neat translation for sums, and sum like operations. The first attempt was made using the Python-Z3 library, an easy to use wrapper for SMT-LIB. The library showed promise in accelerating our development progress but proved to be less than useful due to versioning issues that were not trivially solved.

As a group we decided that it would be more advantageous to continue development with native SMT and discard the previous work done in the Python-Z3 library. We made our first bit of real progress by implementing the axioms outlined in the Leino and Monahan paper [5]. Although we realized a number of the axioms were not applicable to our use-case and decided to take a different development path afterward.

As stated, the first foray into implementation began with declaring the quantifiers as functions in SMT-LIB and then applying the axioms from the [5] paper. Our team began by implementing the simplest of the axioms first. Namely, the induction from above and the base case axioms. Although passable in some aspects, we were unable to apply the axioms to form a complete solution for the implementation of our quantifiers. This being said, the implementation of the [5] axioms was our first step towards success. With this simple solution, we could not guarantee the solver would return SAT when the test cases were incorrect, so we had to look for something different. Now that we had some confidence that we could prove sum-like arithmetic in SMT-LIB, we shifted focus to another promising method of implementation. This being SMT-LIB's recently added recursive syntax.

The newly included recursive syntax in Z3, allowed us to define a function recursively forgoing the inclusion of additional assertions about the function. Our team is however, currently considering including additional axioms to potentially assist in the pitfalls of our current solution. The recursive solution focuses on defining the function recursively within SMT-LIB as follows:

- Base Case: When the current index is less than the minimum of the provided range, return 0 (or 1 for product)
- Determining High/Low

  - Use the filter as a range
  - Extract the low and high bounds from the extremes of the range

- Recursive Step: Accumulate the result of recursion between low inclusively and high exclusively with the value present at low if the filter at high results to true. Otherwise, return the base case value

Below we have an example SMT-LIB translation for a simple sum

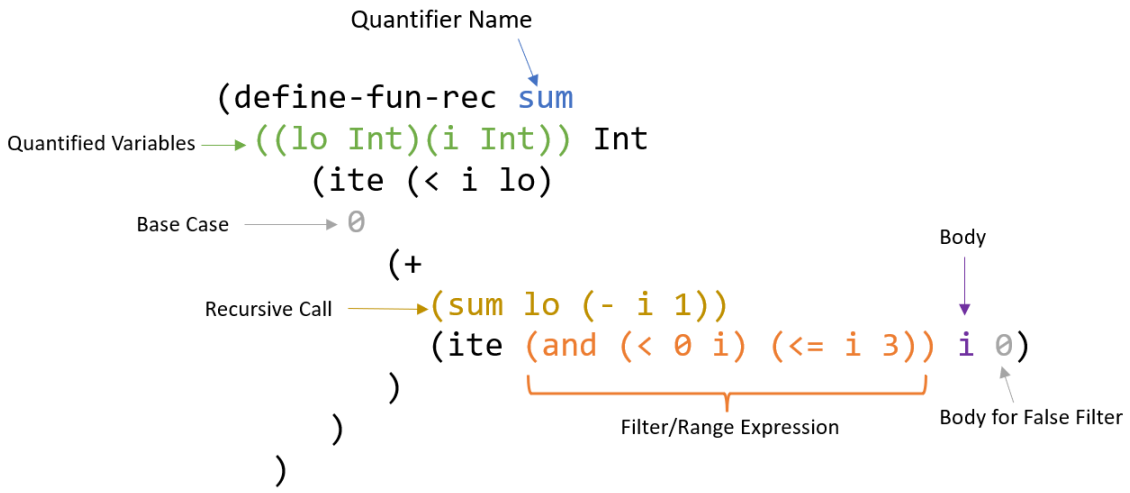


Figure 7: SMT-LIB Translation for a Simple Sum

This solution was far and above our previous solution in terms of correctness. It halted when appropriate and returned UNSAT and SAT correctly for the relevant test-cases. Our group decided we would move forward with the more promising recursive solution.

Utilizing this recursive definition, an algorithm for converting the \sum quantifier was established and from this algorithm all of the other quantifiers followed.

Currently, our solution has accomplished the following:

- Generalized algorithm for converting JML quantifiers into SMT-LIB
- Ability to prove statements containing quantifier expressions
- Complex range handling using the bounds extractor
- Allows for the nesting of quantifier expressions

8.2 Working Relationships

Throughout the researching process, we have kept weekly contact with Dr. Leavens (project sponsor) to discuss developments, issues, and understanding. Dr. Leaven's put our team in direct contact with Dr. Cok whom we met with after establishing a working understanding of the task at hand. Dr. Cok is the primary contributor to the OpenJML project, and his insight was very valuable in understanding the implementation phase of our project.

Our first meeting with Dr. Cok was invaluable as we established an implementation roadmap, and realized the focus of our project was incorrect. Since then correspondence with Dr. Cok has been more regular and with each meeting, we gain more insight into the project and what needs doing.

During the course of our project, we have also reached out to other Senior Design groups that had worked on OpenJML in the past. We established a brief correspondence with the Senior Design group that worked on implementing Floating Point numbers in OpenJML. The Floating Point numbers group primarily assisted us in understanding the OpenJML code-base and navigating its file hierarchy.

8.3 For the Future

As it stands, there are a number of improvements that can be made not only on our solution but in OpenJML as a whole. As OpenJML does not currently cover the entire scope of JML and is in active development, this list of improvements may expand or contract over time.

Our solution although correct, can at times be slow when dealing with large ranges.

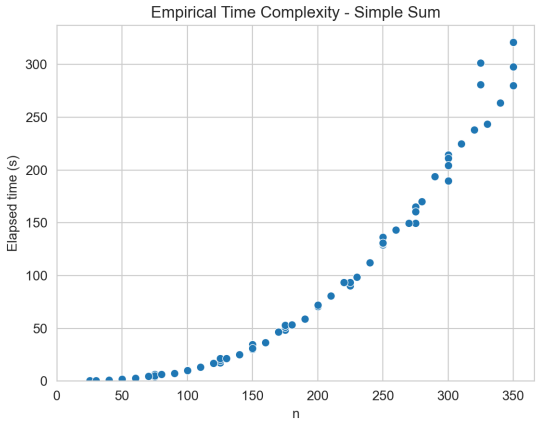


Figure 8: Time Complexity for a Simple Sum (n is upper range bound)

The time complexity appears to be exponential, and indeed when fitting a regression line to the data we see $O(n^{2.54})$ run time. As the range increases, the time it takes to reach a solution grows exponentially.

A number of hypothesis were generated concerning speed improvements. The first of which was shifting the order in which the algorithm progressed through the range. We theorized that the SMT solver would perform less-work when traversing from low to high (rather than descending order). Through another empirical test of the algorithms speed we have confirmed this to be true. This is believed to be as a result of the solver re-using its previous calculations.

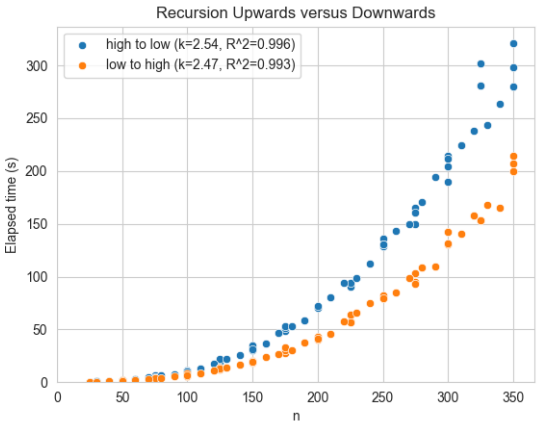


Figure 9: Difference in time-complexity between upwards and downwards recursion

The largest improvement in runtime that we achieved was from implementing the algorithm

in a tail-recursive manner. Meaning the algorithm ends by returning the value of the recursive call. The theory was that removing the caller's frame from the stack should allow additional solver optimization.

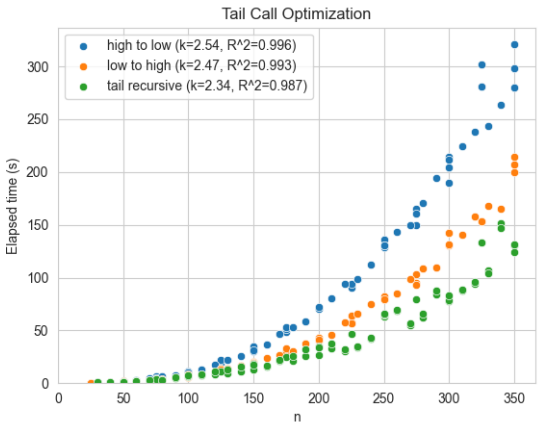


Figure 10: Tail Recursion vs. Other Methods

A focus of future development ought to be improving the translation pipeline to increase the speed of verification when dealing with larger ranges. Our group believes this is as a result of the SMT solver unraveling the function and attempting to solve the proof manually. The current proposed solution is an improved filter function as an additional exploration into this issue.

As a result of our inability to handle large ranges, the current solution cannot handle the infinite ranges from floating point quantified variables. It is natural to assume that working to improve the solution's ability to handle large ranges will apply to this issue as well.

Given the potential for the SMT-Solver to hang when processing our solution on large ranges; it is important to develop a proper warning system for users to ensure they are aware of the possibility of long wait times.

Secondly, the solution is unable to handle the specified syntax for multiple quantified variables. Below we have an example of the appropriate syntax supplied by OpenJML. In this case we have a sum over two variables where i is greater than or equal to zero and i is less than j. We also supply that j must be less than ten.

```
(\sum int i, j; 0 <= i && i < j && j < 10; a[i] < a[j]);
```

Our proposed alternative to achieve this functionality is nesting, an example of which is below. This is not the specified syntax, but achieves the same result.

```
(\sum int i; Range(i); (\sum int j; Range(i, j); Body(i, j));
```

Applying this to our above example

```
(\sum int i; 0 <= i; (\sum int j; i < j && j < 10; a[i] < a[j]));
```

In short, we believe future work ought to be focused on:

1. Understanding why unraveling happens and how it can be prevented
2. Expand warnings around large and infinite ranges. If a solution is devised to appropriately handle large ranges, these warnings will no longer be required
3. Develop support for large/infinite ranges
4. Develop support for floating point quantifier variables
5. Consider the portability of our translations across different solver languages

References

[1] J. Hatcliff et al., "Behavioral interface specification languages," ACM Computing Surveys, vol. 44, no. 3, pp. 1–58, 2012.

[2] G. Leavens et al., "Further lessons from the jml project," 2021.

[3] L. Zedner, "Provable security," 2009.

[4] D. R. Cok and S. Tasiran, "Practical methods for reasoning about java 8's functional programming features," Lecture Notes in Computer Science Verified Software. Theories, Tools, and Experiments, p. 267–278, Nov 2018.

[5] K. R. M. Leino and R. Monahan, "Reasoning about comprehensions with first-order smt solvers," Proceedings of the 2009 ACM symposium on Applied Computing - SAC 09, Mar 2009.

[6] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(¡i¿t¡/i¿)," J. ACM, vol. 53, p. 937–977, nov 2006.

[7] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli, "CVC4," in Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (G. Gopalakrishnan and S. Qadeer, eds.), vol. 6806 of Lecture Notes in Computer Science, pp. 171–177, Springer, 2011.

[8] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB Standard: Version 2.6," tech. rep., Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

[9] L. De Moura and N. Bjorner, "Z3: An efficient smt solver," in Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08/ETAPS'08, (Berlin, Heidelberg), p. 337–340, Springer-Verlag, 2008.

[10] N. Bjorner, C. Wintersteiger, L. Nachmanson, and L. de Moura, "Programming z3."

[11] Y. Cheon, "A runtime assertion checker for the java modeling language," 2014.

[12] J. Gosling, "Class Math," tech. rep., Oracle Corporation, 2021.

[13] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB Standard: Version 2.5," tech. rep., Department of Computer Science, The University of Iowa, 2015. Available at `www.SMT-LIB.org`.

[14] D. Cok, "The SMT-LIB v2 Language and Tools: A Tutorial." `www.SMT-LIB.org`, 2016.

[15] L. de Moura and N. Bjorner, "Z3 - a tutorial."

[16] E. S. L. Lam and I. Cervesato, "Reasoning about set comprehensions," in SMT, 2014.

[17] T. Weber, S. Conchon, D. Déharbe, M. Heizmann, A. Niemetz, and G. Reger, "The SMT competition 2015-2018," J. Satisf. Boolean Model. Comput., vol. 11, no. 1, pp. 221–259, 2019.

[18] A. Niemetz, M. Preiner, A. Reynolds, C. W. Barrett, and C. Tinelli, "Syntax-guided quantifier instantiation," in Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City,

Luxembourg, March 27 - April 1, 2021, Proceedings, Part II (J. F. Groote and K. G. Larsen, eds.), vol. 12652 of Lecture Notes in Computer Science, pp. 145–163, Springer, 2021.