Distributed, Multi-threaded Verification of Java Programs

Perry R. James, Patrice Chalin and Leveda Giannas
Dependable Software Research Group
Department of Computer Science and Software Engineering
Concordia University, Montreal, Canada
{perry,chalin,leveda}@dsrg.org

Abstract

Extended Static Checking (ESC) is a fully automated formal verification technique and is generally quite efficient, as far as verification tools go, but it is still orders of magnitude slower than simple compilation. Verification in ESC is achieved by translating programs and their specifications into verification conditions (VCs). Proof of a VC establishes the correctness of the program. As can be imagined, proving VCs is computationally expensive: While small classes can be verified in seconds, verifying larger programs of 50 KLOC can take hours. To help address this lack of scalability, we present the multi-threaded version of ESC4 and its distributed prover back-end.

1 Introduction

Extended Static Checking (ESC) [14] is a fully automatic form of static analysis that provides more checks than are available from standard type checking but less than from Full Static Program Verification (FSPV) [11]. It does this by translating source code that has been annotated with specifications to Verification Conditions (VCs), which are boolean expressions in first-order logic. If the VC corresponding to a given method can be discharged then the method is correct with respect to its specification. In ESC, VCs are discharged with the help of Automated Theorem Provers (ATPs).

Technology has progressed incredibly since the first ESC tools were developed. We can formally verify non-trivial applications. While small classes can be verified in seconds, larger programs of 50 KLOC can sometimes take hours to verify. We believe that this is an impediment to widespread adoption of ESC: used to modern incremental software development models, developers have come to expect that the compilation (and ESC) cycles are very quick.

In this paper we describe how ESC4 [11] is able alleviate this problem. ESC4 is the ESC component of JML4 [10], a next-generation research platform that provides an Eclipse-based Integrated Verification Environment (IVE) for JML-annotated Java [16]. It offers a full range of verification techniques, including Runtime Assertion Checking (RAC), ESC, and FSPV. More background information is given in Section 2, and an overview of ESC4 is given in Section 3.

ESC4 [11] is able to verify programs faster than its predecessor, ESC/Java2 [12], and other ESC tools such as Boogie [7]. Our contributions are as follows:

- We take advantage of the modular nature of the verification techniques underlying ESC [17] to analyze the methods in a given compilation unit in parallel. This is possible because the ESC analysis done for a given method is independent of that for any others. (Section 4.1)
- We take advantage of ESC4's proof strategies to develop distributed discharging so that non-local resources can be used to reduce the time to verify a set of classes. (Section 4.2)
- The previous two points are achieved by means of OS independent "proof services": If an executable version of a given prover is not available for a given platform, that prover can be exposed through a service and used remotely as if it were local. (Section 4.3)

While tools exist for verifying distributed and multi-threaded code, we have not found another verifier that makes used of these techniques to speed up its own analysis. We believe that ESC4 is the first to do so.

2 Background

ESC4 is a from-scratch rewrite that builds on the lessons learned from earlier projects, principally ESC/Java2. It is part of the JML4 project.

2.1 ESC/Java and ESC/Java2

ESC/Java2 [12] is the successor to the earlier ESC/Java project [14], the first ESC tool for Java. ESC/Java's goal was to provide a fully automated tool to point out common programming errors. The cost of being fully automated and user friendly required that it be–by design–neither sound nor complete. Soundness was lost by not checking for some kinds of errors (e.g., arithmetic overflow of the integral types is not modeled because it would have required what was felt to be an excessive annotation burden on its users). ESC/Java provides a compiler-like interface, but instead of translating the source code to an executable form, it transforms each method in a Java class to a VC that is checked by an ATP. Reported errors indicate potential runtime exceptions or violations of the code's specification. "The front end produces abstract syntax trees (ASTs) as well as a type-specific background predicate for each class whose routines are to be checked. The type-specific background predicate is a formula in first-order logic encoding information about the types and fields that routines in that class use" [14]. The ESC/Java2 project first unified the original program's input language with JML before becoming the platform developed by many research groups.

2.2 JML4

First-generation tools such as ESC/Java and ESC/Java2 are stand-alone command-line applications that use their own custom Java-compiler front ends to produce an AST. Since the research interest of the maintainers of these tools is JML, and not the underlying Java font end, these front ends have not kept up with the latest developments of the Java language.

After much discussion, both within our own research group and with other members of the JML community, it was decided that basing a next-generation JML tooling framework on the Eclipse JDT was the most promising approach.

The result is JML4, a Integrated Verification Environment (IVE) for JML-annotated Java that is built atop the Eclipse Java Development Tooling (JDT).

JML4's first feature set enhanced Eclipse with scanning and parsing of nullity modifiers (nullable and non-null), enforcement of JML's non-null type system (both statically and at runtime) [9] and the ability to read and make use of the extensive JML API library specifications. These include

- recognizing and processing JML syntax inside specially marked comments, both in `.java` files as well as `.jml` files,
- storing JML-specific nodes in an extended AST hierarchy,

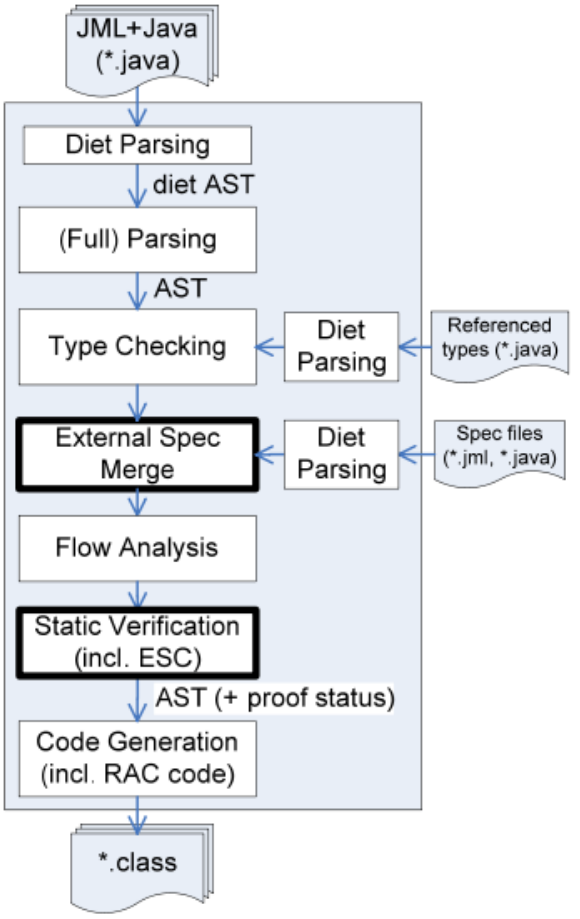

Figure 1. Compiler phases for JML4

- statically enforcing a non-null type system, and
- generating runtime assertion checking (RAC) code.

Since then, work has been underway by several research groups to flesh out JML4 so that it can process all of JML language-level 0 [16].

The framework has also been enhanced to support static analysis [11], including both ESC and Full Static Program Verification (FSPV). The main compiler phases can be seen in Figure 1.

3 Overview of ESC4

ESC4 is the ESC component of JML4 and is a ground-up rewrite of ESC. Its VC generation is based on Barnett and Leino's innovative and improved approach to a weakest-precondition semantics for ESC [8]. One of the most significant results of this approach is that the size of the VCs pro-

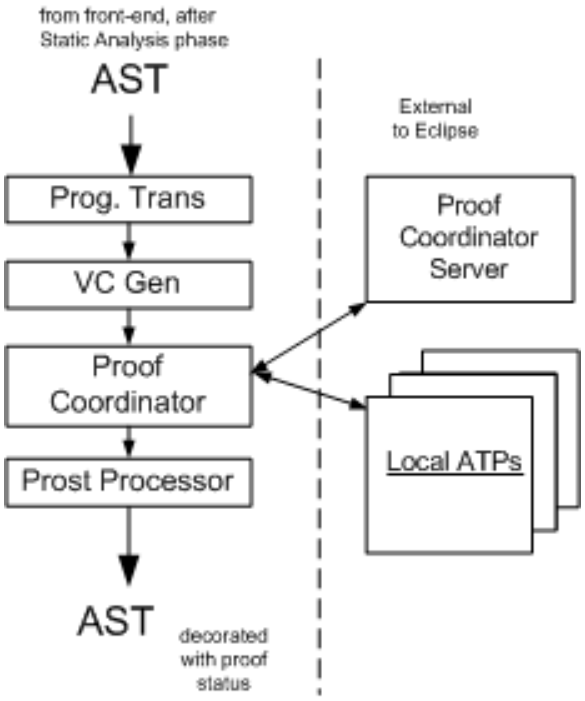

Figure 2. Data flow in ESC4

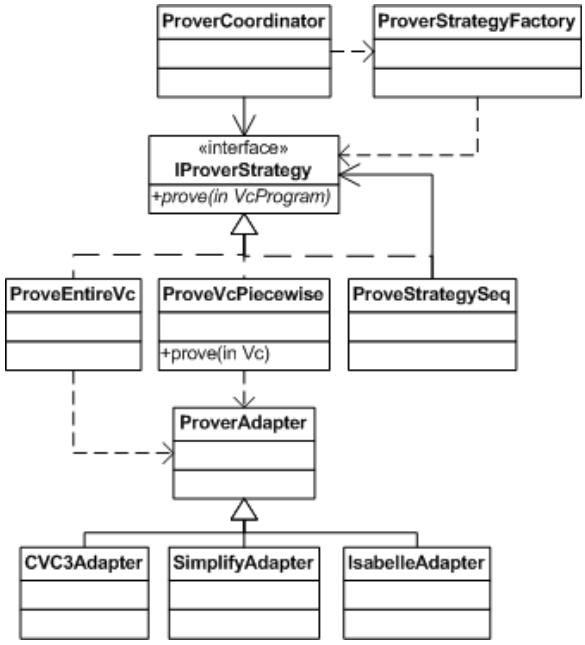

Figure 3. ESC4's prover back-end

duced are linear in the size of the method being analyzed, where earlier approaches generate VCs whose size can be exponential in the worst case.

Figure 2 shows the data flow in ESC4. The fully resolved and analyzed AST produced by the JDT's front end is taken as input. Only those with no front-end-reported errors are processed further by ESC4. The source AST is first converted to a control-flow graph (CFG) as described in [8]. This CFG is similar to Dijkstra's Guarded Command Language [13], except that the guards have been replaced with `assume` statements and the choice operator has been replaced with `goto`s. A VC for each source method is generated from this intermediate form. ESC4's Proof Coordinator is responsible for discharging the VC or reporting why it cannot be discharged. A post-processor reports unprovable VCs to the user through the IVE as failed assertions and attaches the results of the analysis to the original AST. Depending on the compiler options in effect, the code-generation phase may make use of these results to optimize runtime checks.

3.1 Prover back-end

A class diagram for the Prover back-end is shown in Figure 3. A Prover Coordinator is used to discharge VCs. It obtains a proof strategy from a factory whose behavior is governed by compiler options. The default strategy is a sequence of two strategies: The first tries to prove the entire VC using a single ATP. If it fails, the second, `ProveVcPiecewise`, is used. Both use adapters to access the theorem provers. These adapters hide the mechanism used to communicate with the provers. They use visitors to pretty print the VC to produce input for each ATP's native language. To eliminate wasting time re-discharging a previously discharged VC (or sub-VC), the strategies can make use of a VC cache, which is persisted.

`ProveVcPiecewise` implements 2D VC Cascading: VCs are broken down into sub-VCs, giving one axis of this 2D technique, and proofs are attempted for each sub-VC using each of the supported ATPs, giving the second axis.

The conjunction of the set of sub-VCs is equivalent to the original VC. Discharging all of the sub-VCs shows that the method is correct with respect to its specification. Any sub-VCs that cannot be discharged reflect either limitations of the provers or faults in the source.

Currently, three ATPs are used: Simplify, CVC3, and Isabelle/HOL. The first two of these are much faster than Isabelle, but Isabelle is able to discharge VCs containing many constructs that the others are not. After trying both Simplify and CVC3 on a sub-VC, we try to prove its negation before resorting to Isabelle. Only after all other attempts fail is Isabelle invoked.

4 Faster ESC

Applying ESC to industrial-scale applications has been difficult because of the time existing tools require. In this section we highlight the enhancements that have been added to ESC4 that reduce the time needed to verify JML-annotated Java code.

4.1 Multi-threading

Using the arguments in Leino's thesis, *Toward Reliable Modular Programs* [17], it can be shown that each JML-annotated method in a system can be verified independently of the others. Where there are no dependencies, it is easy to introduce concurrency.

First-generation tools such as ESC/Java [14] and ESC/-Java2 [12] were written before multi-threaded and multi-core computers were commonplace. Multi-threading operating systems were already available then, but writing the code to use them would have only increased its complexity without making the processing any faster. This encouraged a serialized approach to the problem, even though the modular nature of ESC is inherently parallelizable. Today, however, multiple-core machines are becoming the norm. Each thread could, in theory, run on its own core and thus reduce the time needed to verify a system to the most needed to verify a single method. While the number of cores needed to achieve this level of speedup will not be available in the foreseeable future, having such small-grained units of work should make efficient scheduling easier for the operating system and/or virtual machine.

Modifying ESC4 to take advantage of ESC's inherent concurrency, of this simply required adding a thread pool: Instead of processing each method sequentially, we packaged the processing (the body of an inner loop) as a work item and added it to the thread pool's task list. Finally, we added a join point to wait until all of the work for a compilation unit's methods finished before ending the ESC phase for it. This last step is necessary because the results of ESC may be used during code generation.

Version 3.4 of the Eclipse Java compiler added the ability to use separate threads to compile individual source files concurrently [3]. Since ESC4 and JML4 are built on top of this compiler, all we had to do to gain this benefit was to ensure that JML4 is thread safe.

The vast majority of the time doing ESC is spent discharging VCs. Specifically, it is the underlying theorem provers that use the most time. For this reason, most ESC tools only make use of a single ATP per verification session. As mentioned above, ESC4 uses 3 by default, and 2D VC Cascading can cause those 3 to be invoked multiple times for each method. Just as the methods in a class can be verified in parallel, the sub-VCs for a method can be discharged in parallel. We just need to put a join point so that we know when the processing of a method's VC has finished.

This gives ESC4 3 layers of parallelism: source files, methods within those files, and sub-VCs for those methods.

4.2 Distributed VC Processing

Once we were able to take advantage of all of the CPU resources on a local machine, it became interesting to ask if we could make use of resources on remote machines. The design of ESC4's Prover Coordinator led to quick discovery of a few deployment scenarios for the distributed discharging of VCs). It was easy to support distributed provers by adding new strategy communication infrastructure.

1. **Prove whole VC remotely**. The first deployment scenario offloads the work of the Prover Coordinator for an entire method. This was done by developing a new subclass of `IProverStrategy` that sends the VC generated for a method to a remote server for processing. (see Figure 4.2). A Proof Coordinator is instantiated on the remote server along with its strategies. We initially had it behave like a local Prover Coordinator and discharge the VC itself with its own local provers.

2. **Prove sub-VCs remotely**. A second deployment scenario was to split the VC into sub-VCs and send each of them off for remote discharging. This was done by extending the `ProveVcPiecewise` strategy discussed in Section 3.1 and having it use remote services to discharge the sub-VCs in parallel.

3. **Doubly Remote Prover Coordinator**. Combining the two previous approaches, so that the remote Prover Coordinator itself delegates the responsibility for discharging the sub-VCs to remote services by using the `ProveVcPiecewiseDistributed` strategy, provides yet another alternative. A deployment view can be seen in Figure 4.2.

Scenario 1 uses the least bandwidth, since only the original VC is transmitted. Scenario 2 uses the next least, although it can be exponentially more than 1. Scenario 3 uses the most, the sum of 1 and 2, but it is split into two groups: the same is used between the the local machine and the remote Prover Coordinator as in 1, and between the remote Prover Coordinator and its servers as in 2.

Splitting a VC into sub-VCs can cause exponential growth in size, since these sub-VCs each represent a single acyclic path from the method's precondition, through its implementation to an assertion.

As a result, scenarios 1 and 3 would be preferred over 2 when the remote machines are not on the same local area network. Scenario 3 can be though of as providing the best

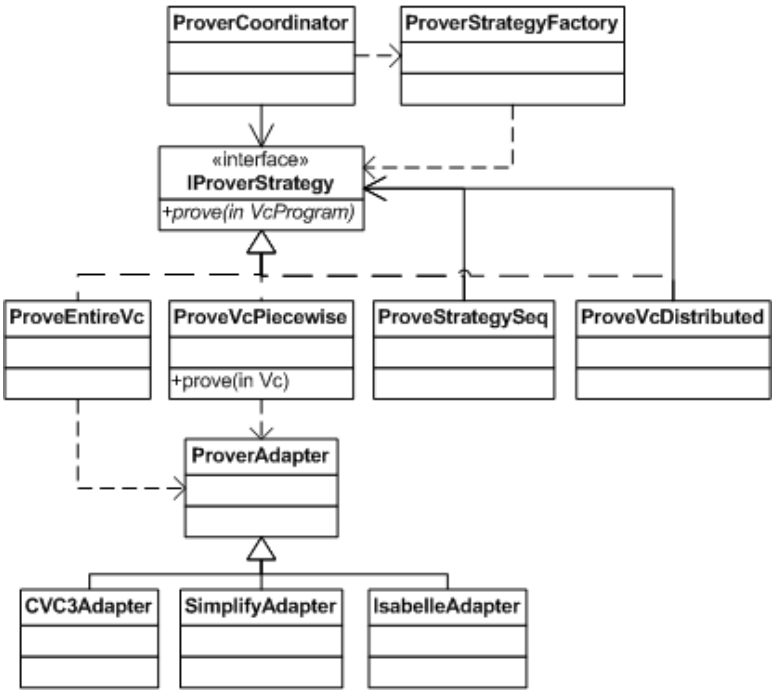

Figure 4. ClassDiagramDistributed

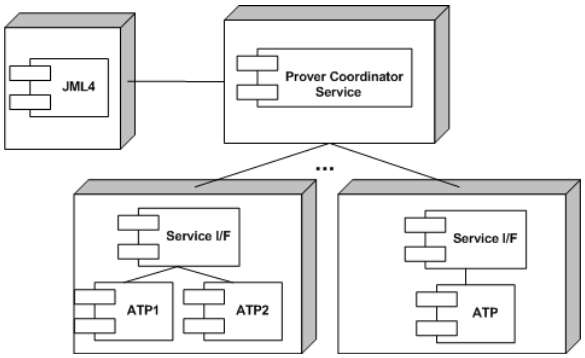

Figure 5. Deployment

parts of the other two: low bandwidth requirements to reach the prover service, and 2D VC Cascading.

In addition, scenario 3 is the most likely to be used when a large farm of servers is available or when the Prover Coordinator service provides a façade that hides load balancing and other details from ESC4.

4.3 Prover service

Independent of the strategy used, the proving resources may be local or remote. The initial prover adapters communicated with local resources using Java's `Process` mechanism. After facing some difficulties installing some provers on all of our development platforms, we hit on the idea of Prover Services.

The adapters that use the provers locally can be taken as base classes to subclasses that access them remotely. Part of the purpose of the adapter classes is to hide the interface with the provers. Applying the same concept lets us hide whether the prover is hosted locally or on a remote machine.

This has the advantage of making the provers OS independent. If a prover is needed on an OS for which there is no executable, it can be hosted on another machine with the appropriate OS and an adapter can hide the extra communication needed to access it.

5 Validation

To confirm that our approach produces speedups, we performed some preliminary timing tests. The source tested was a single Java class with 51 methods. For this code, ESC4 produced 235 VCs. Table 1 shows the number of times each of provers was invoked. Simplify was able to discharge over 80% of the VCs. It was also able to show as false almost 80% of those that were indeed false (23+6). In this sample, CVC3 was not able to prove any of the VCs that Simplify was also unable to prove, and Isabelle was needed for just over 5% of the original VCs.

We ran the test with two deployment scenarios, both

Table 1. VCs discharged with provers

Prover	No. VCs	No. Proved	(%)
Simplify	235	193	82
CVC3	42	-0-	0
Negation[a]	42	23	55
Isabelle	19	13	68
failed	6		

[a]Simplify used to prove the negation of the original VC

Table 2. Timing results

No. servers	No. cores	Time (s) with Prover Coordinator local	remote
1	4	26.6	26.4
2	8	16.9	16.2
3	12	12.8	13.3

based on the Doubly Remote Prover Coordinator described in Section 4.2. In the first, the Prover Coordinator was hosted on the same PC as ESC4. In the second, it was hosted on a faster remote machine.

ESC4 was run on a 2.4 GHz Pentium 4. The Prover Coordinator was hosted either locally to the ESC4 machine or on a 3.0 GHz Pentium 4. Neither of these machines' CPUs is hyperthreaded. The provers were hosted on servers, each with a 2.4 GHz Quad-core Xeon processor. The timing results are shown in Table 5. Each entry in the last two columns is the average of three test runs, which were made after an initial run with the configuration being tested to remove initialization costs. Even so, the timings varied from 0.5 s to 1.6 s. Network usage may account for some of this variation.

For comparison, running the test with the Prover Coordinator and provers were all on the same PC as ESC4 took 72 s. It should be noted that when using remote provers, the CPU of the local machine stayed at 100% during the first few seconds and then dropped to below 20% while gathering the results. When the Prover Coordinator was on a separate machine, that machine's CPU was never went above 50%.

The data gathered indicate that there is little difference between hosting the Prover Coordinator locally or remotely. We had thought that hosting it remotely would allow the VCs to reach the provers faster, thus giving a greater speedup. Surprisingly, as more processing cores were made available, it was actually faster to send the VCs directly. Further testing will have to be done to confirm this. For the sample shown, these timing differences between the two scenarios are within the range of error.

The form of a function from the number of processors used to the time taken to analyze a given piece of code can be derived by applying simple algebra to Amdahl's law [6, 15]. It should have the form

$$t = C_1 + \frac{C_2}{n}$$

Replacing n with 4 and 8 cores and t with the times for the remote Prover Coordinator gives a system of 2 linear equation with 2 unknowns. Solving this system of linear equations gives

$$t = 7.4 + \frac{76.0}{n}$$

The predicted time of 13.7 s for 12 cores is within the error range for experimental value of 13.3 s.

These initial results with up to 12 cores suggest that we should be able to achieve linear speedup by adding more CPUs. One question that future study will have to address is what is the nature of the 7.4 s that cannot be paralellized. After adding 12 cores, this segment takes longer than the portion that is distributed.

6 Related Work

As noted in the introduction, we have not been able to find other existing tools that make use of distributed or parallel processing to enhance program verification. Two related aspects of the work presented here have been previously examined: multi-threaded, distributed compilation and distributed systems in general. These are discussed in the following subsections.

Can we fit the distributed web service here? there's been no documentaion of it.

6.1 Compilation

As mentioned in Section 4.1, Eclipse 3.4 supports mutithreaded compilation of Java programs. The Gnu make command `gmake` has a `--jobs[==n]` option that executes up to n build tasks concurrently. If an integer n is not supplied then as many tasks are started as possible [2]. Microsoft's Visual C++ compiler has the "Build with Multiple Processes" option (`/MP`) that launches multiple compiler processes. If no argument is given, the number of effective processors is used. The number of effective processors is the number of threads that can be executed simultaneously and considers the number of processors, cores per processor and any hyperthreading capabilities.

Several open-source projects and commercial products are available that can distribute the tasks in a build process to networked machines. These only launch a process on a remote machine and do not make use of a SOA approach.

Open-source projects include distcc [4] and Icecream [1]. Xoreax sells a product called IncrediBuild [5] that coordinates distributed builds from within with Microsoft's VisualStudio.

6.2 Distributed services

distributed whatever
-jmeter
they face... not applicable to us

7 Conclusion

Applying ESC to industrial-scale applications has been difficult because of the time required. Invoking a theorem prover for every method in a system is computationally expensive.

We tacked this by applying the "divide and conquer" strategy to allow processing by multiple computing resources, both local and remote. Generating and discharging the VC for Java methods is a problem that can be easily decomposed into many independent tasks. This makes it very amenable to multi-threading and distributed processing.

Given the power of today's desktop PCs, most of an organization's desktop computers' CPUs are under-utilized. Installing a distributed proving service on these machines would allow the organization's developers to tap into existing resources without requiring the acquisition of additional hardware.

The Eclipse JDT compiler is able to process multiple source files in parallel. We showed how we modified ESC4 to support verifying multiple methods in parallel. Similarly, a method's sub-VCs are discharged in parallel. Because of the potential reduction in time to verify a system, it became useful to explore distributed prover resources. This in turn lead to exposing individual provers as distributed resources. All of these combined make the verification of Java programs scalable: The time ESC4 needs to verify a system should be inversely proportional to the CPU resources made available to it.

7.1 Next steps

We modified ESC4 to take advantage of many local and non-local computing resources. The implementation was done to quickly get a usable and stable frame work in place, without much regard for optimization. While we are pleased with the initial results, there are ample opportunities for improvement. These include using more efficient communication mechanisms for communicating with remote resources. Load balancing and other techniques from service-oriented architectures are obvious candidates for consideration.

Proof-status caching, as described in [11], would also improve performance during iterative development since only methods that were changed would need to be re-verified.

After making the obvious enhancements, we plan to conduct timing studies to evaluate the deployment scenarios mentioned in this paper, varying the number and kinds of local and remote resources as well as the characteristics (speed and reliability) of the network.

References

[1] Icecream - openSUSE, 2006. Homepage at http://en.opensuse.org/Icecream.

[2] Parallel - GNU 'make', 2006. Homepage at http://www.gnu.org/software/automake/manual/make/Parallel.html.

[3] Bug 142126 - utilizing multiple CPUs for Java compiler, 2008. Homepage at https://bugs.eclipse.org/bugs/show_bug.cgi?id=142126.

[4] distcc: a fast, free distributed C/C++ compiler, 2008. Homepage at distcc.org.

[5] IncrediBuild by Xoreax Software - Distributed Visual Studio Builds, 2008. Homepage at http://www.xoreax.com/solutions_vs.htm.

[6] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of AFIPS Conference*, pages 79–81, San Francisco, USA, 1967.

[7] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures*, volume 4111 of *LNCS*, pages 364–387. Springer-Verlag, 2006.

[8] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *PASTE '05: The 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 82–87, New York, NY, USA, 2005. ACM Press.

[9] P. Chalin and P. R. James. Non-null references by default in Java: Alleviating the nullity annotation burden. In *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP'07)*, Berlin, Germany, July-August 2007. to appear.

[10] P. Chalin, P. R. James, and G. Karabotsos. An integrated verification environment for JML: Architecture and early results. In *SAVCBS '07: Proceedings of the 2007 conference on Specification and verification of component-based systems*, pages 47–53, 2007.

[11] P. Chalin, P. R. James, and G. Karabotsos. JML4: Towards an industrial grade IVE for Java and next generation research platform for JML. In *VSTTE '08: Proceedings of the 2008 conference on Verifyied Systems: Tools, Techiniques, and Experiments*, 2008.

[12] D. R. Cok and J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *Construction and Analysis of Safe, Secure, and*

Interoperable Smart Devices, volume 3362/2005 of *LNCS*, pages 108–128. Springer Berlin, 2005.

[13] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1976.

[14] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, 2002. ACM Press.

[15] S. Krishnaprasad. Uses and abuses of amdahl's law. *The Journal of Computing in Small Colleges*, 17(2):288–293, 2001.

[16] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. R. Kiniry, and P. Chalin. JML reference manual, 2008. Available at `http://www.jmlspecs.org`.

[17] K. R. M. Leino. *Toward reliable modular programs*. PhD thesis, California Institute of Technology, Pasadena, CA, USA, 1995.