Applications of extended static checking

K. Rustan M. Leino

Compaq Systems Research Center
130 Lytton Ave., Palo Alto, CA 94301, USA
rustan.leino@compaq.com

Abstract. Extended static checking is a powerful program analysis technique. It translates into a logical formula the hypothesis that a given program has some particular desirable properties. The logical formula, called a verification condition, is then checked with an automatic theorem prover. The extended static checking technique has been built into a couple of program checkers. This paper discusses other possible applications of the technique to the problem of producing quality software more quickly.

0 Extended static checking

The use of software plays a large role in our lives, directly and indirectly. We wish it were cheaper to create software, easier to produce software that is correct, and quicker to get to market with new or updated software. A research goal is therefore to improve program quality and programmer productivity.

Many attempts have been made to increase productivity in software development. Perhaps the most successful attempts have been ones that have influenced the design of programming languages. A language can force programs to follow certain programming disciplines. For example, the discipline of storing in each variable values of only one type has been enormously successful. It has led to the development of type systems, type declarations, and type checkers. In fact, type checking has become such a standard occurrence in popular languages that we tend to think of type errors as simple typographic errors rather than thinking about the great damage that such errors might have caused at run time. A type checker is usually applied at the time a program is compiled, which means the errors it catches are found early in the development cycle, which drastically decreases the cost of the errors.

Another example of a programming discipline that has influenced the vast majority of programs being written today is the abstinence from use of arbitrary control-flow jumps. Useful jump disciplines have been incorporated into for example if statements, loops, exceptions, and dynamically dispatched method calls, whereas arbitrary goto statements have ceased to be supported. Because modern high-level languages do not provide constructs for doing arbitrary jumps, the programming errors that stem from arbitrary jumps can be avoided altogether. In this way, good programming disciplines have influenced the design of programming languages, which in turn have a strong influence on the kinds of programs programmers write and the kinds of errors they are likely to introduce.

Another attack on programming errors involves forms of program analysis beyond traditional type checking. The program analysis technique that I focus on in this paper is *extended static checking* (ESC). Drawing from program verification technology, ESC translates a given program (or part of a program) into a logical formula called a *verification condition*. The idea is that the verification condition holds if and only if the program satisfies certain desirable properties. The verification condition is then passed to a theorem prover that searches for counterexamples to the verification condition. Each counterexample corresponds to an error condition in the program.

Two program checkers that use this ESC technique are the extended static checkers for Modula-3 (ESC/Modula-3) [7] and for Java (ESC/Java) [9, 14].

Although extended static checking shares its building blocks with program verification, there's a fundamental difference between the two techniques: the goal of program verification is to prove the full functional correctness of the given program, whereas the goal of ESC is merely to find certain common errors, like null dereferences and array index bounds errors. This difference has far-reaching consequences.

One consequence is that the task of the theorem prover can be expected to be simpler than in program verification. Indeed, in ESC/Modula-3 and ESC/Java, this simplicity has allowed the underlying theorem prover to be entirely automatic. This means that users of ESC-based tools do not need the expertise to drive an interactive mechanical theorem prover; in fact, users don't even need to know about the underlying theorem prover.

Another important consequence of the difference between the goals of ESC and program verification is that the user-supplied specifications can be expected to be simpler. Indeed, in ESC/Java, for example, these light-weight specifications, called *annotations*, revolve around programming concepts and ordinary Java boolean expressions; in contrast, proof strategies and user-defined theories, which are unfamiliar to most programmers, are neither supported nor needed.

A third important consequence of ESC's non-goal of proving the full correctness of the program is that the ESC technique can allow a verification condition generator to be designed to make use of assumptions that are not checked. For example, it may be convenient to assume that no arithmetic overflows occur in the program, or that the object being initialized by a constructor is not dangerously "leaked" to an accessible location before all subclass constructors have completed their initializing actions. If assumptions are not checked, the resulting checking is *unsound*—if the assumptions are violated by the program, then ESC may have missed some of the errors that it was looking for (*cf.* [12]). On the upside, the unsoundness can be chosen by the tool designer to further simplify the task of the theorem prover and to further simplify the annotation language. It is prudent to assume only those conditions that are less likely to be violated in practice.

The level of automation in ESC is reminiscent of type checking. In fact, the name extended static checking comes from the idea that the technique is applied in ways similar to how static type checking is applied, but yielding an extended set of checks. There are more similarities: both static type checking and extended static checking can perform *modular checking*, that is, the checking can proceed module by module, without needing global program information. Modular checking is possible because type checking and ESC rely on annotations that specify module boundaries. For example, a type checker can deal with a call to a procedure P by knowing only the type signature of P (note in particular that the implementation of P is not needed to reason about the call), and the type signature is supplied by the programmer as a type declaration.

ESC annotations are a great feature. Let me list some of their benefits.

One benefit is that annotations allow programmers to write down their design decisions, for example declaring the intention that a certain pointer variable is always non-null or that a certain parameter is expected to be a valid index into a particular array. Not only does writing down the annotations document the design decisions, but it allows ESC to enforce them.

Another benefit is that the modular checking enabled by the annotations can make tools based on ESC scalable. The smaller bodies of code that are analyzed as units permit ESC-based checkers to perform a more detailed analysis. In contrast, whole-program analyses are forced to abstract from the actual code or approximate the behavior of the code in order to scale to large programs. For example, the tool PREfix approximates program behavior by considering only a fixed number of execution paths through each function [3], SLAM abstracts the program behavior by reducing the program's state space to a finite set of predicates [0], and abstract interpretation is a technique for automatically finding an abstraction of the program [4]. While abstraction can be good and is in many situations sufficient to do useful checking, modular checking affords a tool the luxury of using more details.

While annotations have great benefits, the experience with ESC/Java has shown that not everyone embraces the use of annotations. Far from it. The dominant cost in using ESC is the task of supplying annotations. The jury is still out on whether the benefits of ESC at the current required level of annotation outweigh its cost. But regardless of whether the ESC tools built so far are cost effective, one can apply the ESC technique differently from how it is applied in the checkers ESC/Modula-3 and ESC/Java. Different applications of ESC can adjust the required level of annotation differently and can maybe even change how annotations are perceived. In the rest of this paper, I discuss four such possible applications.

1 New applications of the ESC technique

1.0 Changing the unit of modularity

The checkers ESC/Modula-3 and ESC/Java perform modular checking at the level of separate routines. That is, these checkers analyze the implementation of one routine using only the specifications, not the implementations, of routines that are called. This approach follows the ideas behind procedural abstraction in that the specifications explicitly establish a contract between callers and the implementation of each routine. Because programmers tend to divide their code into reasonably-sized routines to limit the complexity of each routine, the approach of using routines as the unit of modularity naturally imposes comfortable limits on the amount of code that is analyzed at a time.

However, sometimes this level of granularity may not be ideally suited. For example, in object-oriented languages like Java, the program checker can often use the code of a private method in lieu of (or in addition to) a specification of the method. This suggests that one could use the class, rather than the routine, as the unit of modularity. By enlarging the unit of modularity in this way, this approach would reduce the annotation burden. In Java, one can even consider going a step further, making the package (which is a set of classes and interfaces) be the unit of modularity.

While enlarging the unit of modularity has the benefit that less annotation is needed, the cost is that verification conditions grow bigger, and the number of execution paths to consider through the code can grow dramatically.

So, new applications of ESC can reduce the annotation burden by adopting a different unit of modularity. This is similar to the approach taken by the type system in ML, where type inference reduces the need for manual type annotations on every function [16].

1.1 Using ESC as a subroutine

Another promising application of the ESC technique is as a precise subroutine for analyzing a small piece of code in the context of a more encompassing checker. Let me mention two current efforts in this application area.

Houdini is a program checker that first infers annotations for a given (unannotated) program and then invokes ESC/Java on the annotated program to find possible errors [10]. The inferred annotations reduce the number of spurious warnings produced by the call to ESC/Java. Houdini also makes use of ESC/Java in its inference algorithm: it iteratively calls ESC/Java as a subroutine in order to discover annotations that are consistent with the program.

Another ongoing research effort is to use the tool Daikon's run-time profiling and statistical invariant inference [8] to create annotations that are then checked by ESC/Java [18]. By using ESC as a subroutine, the inferred likely invariants can be checked for all program inputs, not just the inputs used during the run-time profiling.

1.2 Annotation checkers

Another application of the ESC technique can be found by applying a tool like ESC/Java in a particular, limited way. By default, ESC/Java warns about the possible misapplication of language primitives (like null dereferences and array bounds errors) whenever it cannot establish that such errors won't occur. For an unannotated program, this leads to a large number of warnings, which is not useful to programmers.

An alternative is to permanently turn these warnings off (which ESC/Java's command-line switches permit), letting the tool check only that the program is consistent with the given annotations. The resulting tool, an *annotation checker*, will then produce no warnings on an unannotated program. However, as soon as an annotation is added, the checker goes to work. This mode of checking, although it won't find checked run-time errors, lets programming teams ease their way into using the ESC checker. Even though annotations are required in order for the tool to do any actual checking, the annotation burden that otherwise arises from checking the use of language primitives is avoided. Stated differently, users get only what they pay for.

It is worth pointing out a particular specification paradigm that seems ideally suited for an annotation checker. The paradigm specifies a protocol that clients of a given class must satisfy. For example, consider a class representing character input streams and containing the operations *open*, *getChar*, *unGetChar*, and *close*. Suppose the intended use of these operations is as follows: *open* must be applied before any other operations are applied, no operations are allowed after applying *close*, and *unGetChar* is allowed only if the previous operation applied to the stream was *getChar*. To specify this protocol, we can introduce two boolean object fields *isOpen* and *isUnGettable* (ESC/Java allows the declaration of *ghost fields* for when the fields don't need to be seen by the compiler) and specify the four methods as follows:

/*@ **modifies** *isOpen*, *isUnGettable*; **ensures** *isOpen* */
void *open*(*String filename*);

/*@ **requires** *isOpen*; **modifies** *isUnGettable*; **ensures** *isUnGettable* */
int *getChar*( );

/*@ **requires** *isOpen* & *isUnGettable*; **modifies** *isUnGettable*; */
void *unGetChar*( );

/*@ **requires** *isOpen*; **modifies** *isOpen*, *isUnGettable*; */
void *close*( );

where **requires** specifies a method's precondition, **modifies** specifies which variables the method may modify, and **ensures** specifies the method's postcondition.

In summary, simple protocols like the one above can easily be specified in annotations. An annotation checker then checks that any specified protocol is observed by clients. While one can write protocol specifications like this one using special languages for abstract state machines (see *e.g.* AsmL [1] and Vault [5]), the ESC technique can perform a detailed semantical analysis and allows such protocol specifications to co-exist with other kinds of specifications.

1.3 Programming language design

As history shows, programming language design, for those languages that attract users, is one of the most effective ways to influence software development practices. The application of the ESC technique in programming language design seems intriguing, notably in cases where the analysis to be performed is mostly local to a routine or when the annotation overhead is small. I will outline three specific example applications, and will then discuss some obstacles that must be overcome to make these applications a reality.

A primary goal of programming language design is to find grammatical restrictions, static checks, and an execution semantics that together guarantee certain invariants at run time. For example, static and dynamic type checks can guarantee the invariant that variables hold values of their declared types. One can take the type systems of today's popular languages a step further by distinguishing between non-null types and may-be-null types. For example, let the type $T-$ represent the set of objects of a class T (or of a subclass of T), and let the type $T+$ indicate the union of that set and the special object **null**. A dynamic type cast of an expression e from $T+$ to $T-$ gives rise to a run-time check that e does not evaluate to the value **null**. The languages Java, Modula-3, and C++ provide only type $T+$, whereas the language CLU (see *e.g.* [15]) provides the type $T-$ and allows type $T+$ to be defined using a tagged union. Perhaps a reason for why this null-value discrimination in types has not yet found its way into more languages is that it is awkward (in the language definition and in programs) to have to have a special construct for breaking up the tagged union into two cases. More natural may be simply to use an ordinary if statement and compare the value of a $T+$ expression with **null** before using the expression as a $T-$ expression. The ESC technique may see a useful application here, because it could do a precise analysis of each routine implementation, producing an error for any possible dereference of **null** or possible assignment of **null** to a $T-$ variable. That is, the language would sport both types $T-$ and $T+$, but would require programmers to insert a dynamic type cast from $T+$ to $T-$ only when ESC would otherwise have produced an error.

Let's consider a second example. Uninitialized local variables is a problem in some languages, like C, and lint-like checkers are wise to warn about situations when this is a possible error. To prevent such errors, Java builds in a "definite assignment" rule that guarantees that every local variable is initialized (to a value of the variable's type) before it is used. The rule usually does well, but not for a code segment like

$$
\begin{array}{l}
\textbf{int}\ x; \\
\textbf{if}\ (B)\ \{\ \ldots\text{assign to}\ x\ldots\ \} \\
\ldots \\
\textbf{if}\ (C)\ \{\ \ldots\text{use}\ x\ldots\ \}
\end{array}
$$

where condition C evaluates to true only if the first if statement's then clause is executed. The coarseness of Java's definite assignment rule will cause the compiler to report an error for this code segment (likewise, any coarse lint-like checker will produce a spurious warning). In these cases, one might have wished for a dynamic check instead, but programmers are forced to add a "dummy" assignment to x before the first if statement. If the programmer's assumption about condition C is mistaken, then the dummy assignment will have introduced an error that is harder to catch. Here, the more detailed analysis of the ESC technique can come to the rescue (see *e.g.* ESC/Java's **uninitialized** annotation [14]).

As a third example, we may consider including language support for specifying and checking protocols like those referred to in the previous subsection.

To realize any of these improvements in language design through applications of the ESC technique, one must overcome some obstacles. I'll describe the three primary obstacles.

A first obstacle is that ESC checking can be unsound. For a language to guarantee certain program invariants, the ESC technique had better be applied only in sound ways. One may be able to achieve soundness, but the challenge is to do so while retaining the power of the ESC technique and keeping annotation to a bare minimum.

Another obstacle is the possible need to know which variables a routine can modify. For instance, to enforce a protocol like the one in the example in Section 1.2, one needs to know whether or not a call can modify *isOpen*. Specifying the possible modifications of a routine is difficult, because it adds complexity to the annotation language [11, 13, 17]. In fact, to ensure soundness, one may need to introduce a discipline for alias confinement [6], which may in turn require knowing which variables a routine can read [2].

A third obstacle is that to include ESC checks among the static checks prescribed by a programming language, the language definition must include enough of a description of the verification condition generator and theorem prover that all compilers will agree on which programs are to be accepted. For example, if the language definition were to leave the strength of the theorem prover unspecified, then a program accepted by a compiler with a powerful theorem prover might not be accepted by a compiler with a more impoverished theorem prover. There is hope, however: a language may define the generation of verification conditions into some decidable mathematical domain. For example, if the verification conditions are formulas in first-order logic with uninterpreted function symbols and equality, then a compiler can use any sound and complete decision procedure for congruence closure—the language definition would cleanly say what needs to be checked, not how it is to be checked. Restricting verification conditions to decidable domains limits the power of the ESC technique (*cf.* [12]), but in the case of language design, such limitations may be appropriate.

2 Conclusion

Extended static checking is a powerful program analysis technique. To date, the primary application of the technique has been to standalone program checkers that search for misapplications of language primitives, perform modular checking routine by routine, and require annotations. This paper has sketched some new application areas of the ESC technique. One of these areas for which the potential gain is particularly high is programming language design, but several research challenges remain to make this application a reality. By exploring various applications of the ESC technique, possibly in combination with other techniques, we can hope to reduce the cost in producing quality software.

Acknowledgements

I am grateful to Cormac Flanagan, Raymie Stata, and Marcos Aguilera for contributing ideas and feedback in various discussions on the subject of this paper, and to Tony Hoare for articulating in an inspiring way some important correlations between programming language design and program correctness.

References

0. Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings SPIN 2001*, 2001. To appear.
1. Mike Barnett, Egon Börger, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Using abstract state machines at Microsoft: A case study. In *Abstract State Machines, Theory and Applications, International Workshop, ASM 2000*, volume 1912 of *Lecture Notes in Computer Science*, pages 367–379. Springer, 2000.
2. John Boyland. Alias burying: Unique variables without destructive reads. *Software—Practice & Experience*, To appear.
3. William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software—Practice & Experience*, 30:775–802, 2000.
4. Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM, January 1977.
5. Rob DeLine and Manuel Fähndrich. Vault project home page. On the web at `http://research.microsoft.com/vault/`, 2001.
6. David L. Detlefs, K. Rustan M. Leino, and Greg Nelson. Wrestling with rep exposure. Research Report 156, Digital Equipment Corporation Systems Research Center, July 1998.
7. David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, December 1998.
8. Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000), Limerick, Ireland*, June 2000.
9. Extended Static Checking for Java home page. On the web at `http://research.compaq.com/SRC/esc/`, November 2000.
10. Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In José Nuno Oliveira and Pamela Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity*, volume 2021 of *Lecture Notes in Computer Science*, pages 500–517. Springer, March 2001.

11. K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '98)*, volume 33, number 10 in *SIGPLAN Notices*, pages 144–153. ACM, October 1998.
12. K. Rustan M. Leino. Extended static checking: A ten-year perspective. In Reinhard Wilhelm, editor, *Informatics—10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 157–175. Springer, January 2001.
13. K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. Research Report 160, Compaq Systems Research Center, November 2000.
14. K. Rustan M. Leino, Greg Nelson, and James B. Saxe. ESC/Java user's manual. Technical Note 2000-002, Compaq Systems Research Center, November 2000.
15. Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Electrical Engineering and Computer Science Series. MIT Press, 1986.
16. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.
17. Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, FernUniversität Hagen, 2001. Available from `http://www.informatik.fernuni-hagen.de/pi5/publications.html`.
18. Jeremy W. Nimmer and Michael D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In preparation, MIT, 2001.