Extended Static Checking: a Ten-Year Perspective

K. Rustan M. Leino

Compaq Systems Research Center
130 Lytton Ave., Palo Alto, CA 94301, USA
rustan.leino@compaq.com
http://research.compaq.com/SRC/personal/rustan/

Abstract. A powerful approach to finding errors in computer software is to translate a given program into a verification condition, a logical formula that is valid if and only if the program is free of the classes of errors under consideration. Finding errors in the program is then done by mechanically searching for counterexamples to the verification condition. This paper gives an overview of the technology that goes into such program checkers, reports on some of the progress and lessons learned in the past ten years, and identifies some remaining challenges.

0 Introduction

Software plays an increasingly important role in everyday life. We'd like software to be reliable, free of errors. The later an error is found, the more expensive it is to correct. Thus, we would like to detect software errors as early as possible in the software design process. Static program checkers analyze programs in search of errors and can be applied as soon as program development commences.

Many kinds of static program checkers are possible. To facilitate comparison between these from a user's perspective, it is useful to assess checkers along two dimensions, *coverage* and *effort*. The coverage dimension measures the proportion of errors in a program that are detected by the checker, giving a sense of what a user may expect to get out of the checker. The effort dimension measures how arduous it is to put the checker to use, giving a sense of what a user has to put in to benefit from the checker. Factors that contribute to the effort dimension include the time spent learning to use the checker, preparing a program to be input to the checker, waiting for the checker to complete, deciphering the checker's output, and identifying and suppressing spurious warnings.

Figure 0 shows some classes of static checkers along the two dimensions. In the lower left corner of the figure, depicting low coverage at low effort, we find checkers like type checkers and `lint`-like [17] checkers. Many programmers use checkers like these, because they perceive the benefit as outweighing the effort. The overall coverage may be low, but the kinds of errors caught by these checkers are common and relatively cheap to find.

In the upper right corner of the figure, depicting high coverage at high effort, we find full functional program verification. Here, the coverage approaches 100

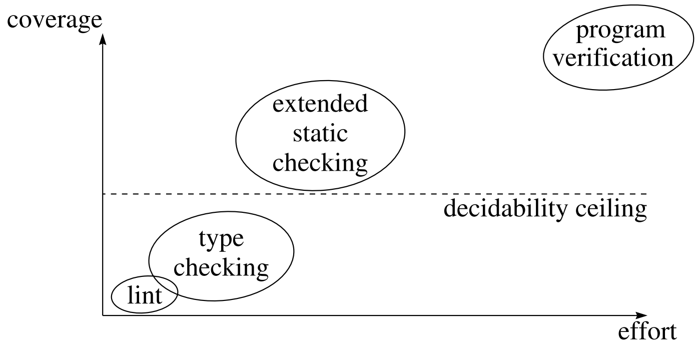

Fig. 0. Some classes of static checkers plotted along the two dimensions coverage and effort. The illustration is not to scale.

percent, but the effort required is tremendous, usually including tasks such as formalizing every detail of the program's desired behavior, axiomatizing mathematical theories that are often quite subtle, and hand-guiding a theorem prover through the proofs of correctness. Consequently, only for a small number of application areas, where programs are very small or where the cost of a software error could be devastating, does full functional program verification stand a chance of being cost effective.

Figure 0 includes a horizontal line labeled *decidability ceiling*. This is a limit along the coverage dimension below which the checker technology can provide certain mathematical guarantees. For example, the techniques applied below the decidability ceiling may run in, say, linear or cubic time, whereas the techniques required to achieve coverage above the decidability ceiling may have infeasible worst-time running times or may not even be decidable.

But giving up on the guarantees provided below the decidability ceiling may not be so bad in practice. By aiming to ascend above the decidability ceiling, one can explore uses of more powerful technology, hoping to find uses that will be reasonable for most of the programs given as input to the checker.

This paper focuses on a particular class of checker, an *extended static checker*, which (see Figure 1) analyzes a given program by generating *verification conditions*, logical formulas that, ideally, are valid if and only if the program is free of the kinds of errors under consideration, and passes these verification conditions to an automatic theorem prover which searches for counterexamples. Any counterexample context (predicate that, as far as the theorem prover can determine, is consistent and implies the negation of the verification condition) reported by the theorem prover is translated into a warning message that the programmer can understand. Extended static checking lies above the decidability ceiling in Figure 0, because it provides better coverage than traditional static checkers can achieve; and it lies way to the left of full functional program verification on the effort dimension, because the effort required to use it is considerably smaller.

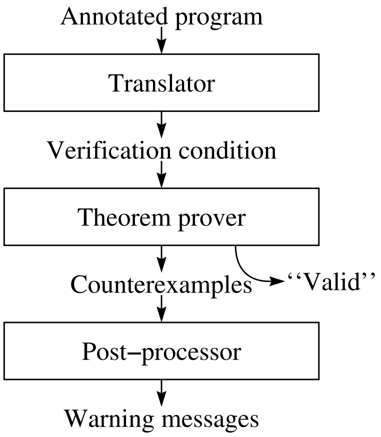

Fig. 1. Extended static checker architecture.

Early work with similar goals includes Dick Sites's PhD thesis [37] and Steve German's Runcheck verifier [15].

The present paper draws on the experience with building and using two extended static checkers during the last decade, both at the Compaq Systems Research Center (which belonged to Digital Equipment Corporation for most of that decade). Starting with some early experiments in 1991, the Extended Static Checking for Modula-3 (ESC/Modula-3) project developed the underlying checking technology in the years that followed [9]. By 1996, the checker proved usable for a number of systems modules, but by that time the pool of Modula-3 programmers in the world had dried up. In 1997, the Extended Static Checking for Java (ESC/Java) project began, seeking to build an extended static checker that would appeal to a large number of programmers [12, 22]. ESC/Java adapted the technology developed for ESC/Modula-3, and made some significant changes in the annotation language in order to make the checker easier to use. Both of these checkers have been applied to thousands or tens of thousands of lines of code, and both have found errors in programs with many users.

Many research challenges were encountered during these two projects. For example: Which errors should the checker check for? Is there a suitable semantics for real (non-toy) programming languages? Can the theorem prover be automatic and fast enough? Can the theorem prover produce counterexample contexts, and can these be turned into useful warning messages? Is annotation of programs possible and can it avoid being onerous? This paper addresses these challenges and reports on the progress made toward overcoming the challenges as part of the two extended static checking projects. The paper then sketches some future challenges and possible research enterprises in this area of program checking for the next decade.

1 Challenges

In this section, I describe five major research challenges faced in the course of the two extended static checking projects and report on the status of our efforts to overcome these challenges.

1.0 Deciding which errors to check for

A first research challenge in designing any kind of program checker is deciding what the checker is going to be good for. That is, what kinds of programming errors should the checker search for? This is an important decision, because some errors may occur more frequently than others, may have more disastrous effects than others, or may be more difficult than others to find using existing program checkers.

But this basic question immediately leads to another fundamental question that will no doubt send shivers down the spine of the purist: *So what about soundness?* Using the terminology of program verification, a checker is *sound* if it does not miss any errors in the program. Thus, deciding about which errors to check for means deciding how sound the checker should be. Why would we ever want to give up on soundness? Because soundness affects the complexity of the annotation language and the annotation burden, both of which contribute to the effort dimension of the checker.

Let me give an example. Suppose we were to check programs for arithmetic overflows. Precisely tracking the possible values of integer variables in all possible executions of a program is infeasibly difficult. The situation can be alleviated by using approximations [7] or relying on the programmer to supply hints in annotations. But the reason a particular integer operation does not overflow can easily be quite complicated. If the approximation machinery is not up to the task, the checker will produce many spurious warnings, which increases the effort in using the checker. Similarly, it takes an advanced annotation language to allow, say, writing down the precondition that guarantees that a matrix multiplication routine does not cause an overflow. The learning curve for such a language and the burden of actually annotating a program with the pertinent properties increase the effort in using the checker. And after all that effort, does, say, a caller of the matrix multiplication routine stand any chance of actually discharging the precondition? Is this effort worthwhile?

Even if we could do the analysis perfectly for arithmetic overflow, it's not certain that the result would be desirable in many applications. In some programs, it is conceivable that a non-negligible fraction of arithmetic operations may indeed overflow in certain runs of the program where the input is orders of magnitude larger than anticipated by the program designers and programmers. Then the many warnings produced by the checker may not be spurious after all, but the programmer would still have no interest in changing the program to properly handle such inputs—a sentence in the program's user manual would both be appropriate and require much less effort.

This example illustrates various design decisions that the designer of a program checker wrestles with daily. By introducing unsoundness in just the right places, the checker can achieve a better position in the coverage-to-effort design space.

We designed ESC/Modula-3 and ESC/Java to check for errors in three categories. First, they check for conditions for which the languages Modula-3 and Java prescribe run-time checks: null dereferences, array index bounds errors, type cast errors, division by zero, etc. Second, they are capable of checking for common synchronization errors: race conditions and deadlocks. Third, they check for violations of program annotations: for example, the checkers warn if a call site fails to meet a declared precondition or if a routine body fails to maintain a declared invariant.

Users can disable checking for any of the kinds of errors on a line-by-line basis by using a **nowarn** annotation or globally by using a command-line switch.

The checkers were designed not to look for problems with non-termination (because of the difficulty in providing appropriate annotations, and because some programs are designed to run continuously), arithmetic overflow, out of memory conditions (because of the flood of warnings of doubtful utility that this would produce), and various kinds of "leaking" and "rep exposure" problems (see, *e.g.*, [8, 24, 22]). One of the changes from ESC/Modula-3 to ESC/Java is that ESC/Java does not enforce **modifies** clauses (specifications that limit which variables a routine is allowed to modify), because of the requirements that **modifies** checking places on the annotation language [20, 21] and on the user of the checker.

One final note about designing which errors to check for: by not checking for one kind of error, other errors may be masked. For example, consider the following (Java) program fragment:

$$
\begin{array}{l}
\textbf{if}\ (0 \leq x\ \ \&\&\ \ 0 \leq y)\ \{ \\
\quad \textbf{int}\ z = x + y\,; \\
\quad \textbf{int}[\,]\ a = \textbf{new}\ \textbf{int}[z]\,; \\
\quad \vdots \\
\}
\end{array}
$$

The call to **new** allocates an integer array of size z, so z is required to be non-negative. In ordinary mathematics, this follows from the fact that z is the sum of two non-negative integers, but due to the possibility of arithmetic overflow (or addition modulo 2^{32}, which is what Java uses), z may actually be negative. Thus, even if the checker generally looks for negative-size array allocation errors, it may miss some such errors by not considering arithmetic overflow.

1.1 Defining formal semantics for modern languages

To build a static program checker that is capable of finding semantic errors in programs, one needs a formal semantics for the source language. Through the formal semantics, one can translate a given program into a set of verification conditions. But modern programming language like Modula-3 and Java include not just loops and procedures (which sometimes are considered difficult in their own right), but also object-oriented features like objects (which are references to data records and method suites), subtypes, dynamically dispatched methods, information hiding, and even concurrency features! Are these not difficult, or impossible, to model?

Indeed, if these features were to be used in unrestricted ways, untangling the mess into a formal semantics appears to be an impossible task [4]. Luckily, good programmers impose a structure on the way they use the features of the language; they follow a *programming methodology.* This helps them manage the complexity of their programs. Not only can a program checker take advantage of the fact that programmers use methodology, but it may also be a good idea for the program checker to enforce certain parts of the methodology.

The general concept of using *specifications* is fundamental to good programming methodologies. A specification is a contract that spells out how certain variables, procedures, or other constructs are to be used within the program. Once they are part of the source language, specifications help define the semantics of the language. For example, by using the programming methodology of associating a specification with every procedure, the formal semantics of a procedure call can be defined in terms of the procedure's specification alone, independent of the procedure's implementation. When making use of such a methodology, it seems prudent also to enforce the methodology, which is done by checking that every procedure implementation meets its specification. This old and fundamental idea [33] too often seems to be forgotten.

Making use of programming methodology overcomes the impossibility of designing a formal semantics, but the task may still seem unwieldy. To manage this complexity, we have found it convenient to translate the source language into a small intermediate language whose formal semantics is easy to define. Such a translation task is comparable to the compiler task of translating a source language into a more primitive intermediate language (like three-address codes [0]). We have used a variation of Dijkstra's guarded commands [10] as our intermediate language [23].

An intermediate language is good at capturing the essence of executable code in the source language, but may not be well suited for capturing all important information in the source language, especially if the intermediate language lacks declarations and types. We have found that we can encode the remaining information into a logical formula that we refer to as the *background predicate.* The background predicate, which is used as an antecedent of the verification condition, formalizes properties of the source language's expression operators and type system. (For a full description of the background predicate of a small object-oriented language, see Ecstatic [19].)

By giving a small example, I will attempt to convey a flavor of the translation of source-language programs into verification conditions. Consider the following Java class declaration:

$$\textbf{class } T \textbf{ extends } S \; \{ \; \ldots \; \}$$

which introduces T as a subclass of S, and consider the following Java program fragment:

$$t = (T)s\,;$$

where t is a variable of static type T and s is a variable of static type S, and where the Java type-cast expression "$(T)s$" returns the value of s after checking that this value is assignable to type T. The background predicate includes a relation $<:_1$ ("direct subtype") on class names, and the Java class declaration above contributes the following part of the definition of that relation:

$$T <:_1 S$$

The background predicate defines a relation $<:$ as the reflexive transitive closure of $<:_1$. It also includes a predicate is, where $is(o, U)$ means that the value o is assignable to type U. This predicate is defined as follows:

$$(\forall\, o, U \;::\; is(o, U) \;\equiv\; o = \mathbf{null} \vee \mathit{typeof}(o) <: U\;)$$

where *typeof* maps non-null objects to their (dynamic) types. The translation of the assignment statement in the Java program fragment above produces the intermediate-language command

$$\mathbf{assert}\; is(s, T)\;;\;\; t = s$$

That is, before the actual assignment of s to t, the command explicitly checks that the value of s is assignable to type T. After applying the semantics of the intermediate language, for example using *weakest preconditions* [10], the verification condition takes the shape

$$\ldots T <:_1 S \wedge (\forall\, o, U \;::\; is(o, U) \equiv \ldots)\ldots \;\Rightarrow\; \ldots is(s, T) \wedge \ldots$$

This illustrates that one needs to prove $is(s, T)$ from the background predicate and from what is known about s.

As alluded to above, ESC/Modula-3 and ESC/Java reason about each call in terms of the specification of the callee. This allows the checkers to perform *modular checking*, which means that to check one module (or class) M, the checker only needs the declarations and specifications in the modules (or classes) that M imports (that is, builds on or uses). In particular, the implementations of the imported modules (or classes) are not needed. Consequently, one can check code that calls into a library module whose implementation details are hidden, and one can check the uses and implementation of a class without needing all its future subclasses. Modular checking is an important asset of the checkers; unfortunately, as we shall see later, it is also a liability.

1.2 Using a theorem prover

Once a verification condition has been produced, the next checker task is to attempt to find counterexamples to it. In many ways, the structure of verification conditions mimics that of the statement and expression constructs of the program. Typical verification conditions thus include many cases to be checked, but each case tends to be mathematically shallow. This makes the task ideally suited for a mechanical theorem prover.

The desire to keep the effort of using the program checker low places three requirements on the theorem prover. First, the theorem prover must run entirely automatically, with no user interaction. Any degree of user interactivity would mean that users of the program checker would need to learn to operate the theorem prover, something which takes a lot of training. Second, the output of the theorem prover needs to include a list of counterexample contexts. When a program contains errors and the verification condition is invalid, it would be unacceptable if the theorem prover's failure to prove the verification condition were not accompanied with a reason for the failure. Third, the theorem prover must be reasonably fast, because any program checker that is to be part of the program development process will need to be run frequently. Can these three requirements be met?

In our experience, we have found that it is indeed feasible to use a theorem prover in a program checker to do semantic analysis. Our theorem prover, called *Simplify*, works on the kinds of formulas that our extended static checkers produce as verification conditions, is entirely automatic, outputs counterexample contexts, and usually performs well. We have used the same theorem prover for both ESC/Modula-3 and ESC/Java.

The theorem prover Simplify is based on the Nelson-Oppen algorithm for cooperating decision procedures [30, 31]. Simplify's decision procedures include an Egraph for the theory of equality including congruence closure, a simplex solver for linear arithmetic, a backtracking search for disjunctions, a matcher for universally quantified formulas, and an ordering-theory procedure for various partial orders. When Simplify was built as part of the ESC/Modula-3 project, we developed various heuristics that make it work well for the kinds of formulas that arise as verification conditions. We have since added some theorem prover features that have allowed us to enhance the output of ESC/Java, but the built-in heuristics have remained the same as they were for ESC/Modula-3.

There are many mathematically equivalent ways in which one can formulate verification conditions. Alternative formulations may differ dramatically in how they impact the theorem prover's performance. In building an extended static checker, one must pay ample attention to crafting the verification conditions carefully, and doing so requires both expertise with and experimentation with the underlying theorem prover.

An important desideratum of a program checker is that it not get stuck for too long (or forever!) in trying to find errors in some part of the given program. Consequently, we impose a time limit for each verification condition in ESC/Java (the default is 5 minutes, which is occasionally reached).

In Section 1.0, I emphasized the importance of allowing unsoundness into the design space of program checkers. However, unsoundness should be confined to areas that can be explained in the checker's user manual, so that programmers can understand the limitations. This suggests that it is prudent not to design unsoundness into the underlying theorem prover, because it may be hard to predict where such unsoundness may strike and to explain what the programmer can do to avoid the unsoundness.

1.3 Producing meaningful warning messages

The next challenge is to turn verification-condition counterexample contexts, as output by the theorem prover, into meaningful warning messages. Users of the program checker should not be required to grok the theorem prover or the particular encoding of verification conditions in order to understand the checker's warning messages.

Because the structure of verification conditions mimics the structure of the program, particular parts of the verification condition correspond to checks being made at particular points in the program. By tracking which parts of a verification condition the theorem prover uses in the counterexample contexts it reports, we have found that one can accurately recover the kind of error (null dereference, array index bounds errors, etc.) and source location of the error, even an execution trace leading to the error, from the theorem prover's output.

Tracking the parts of the verification condition that the theorem prover is currently considering can be done in several ways. One effective way involves a theorem-prover *labeling* mechanism. The theorem prover Simplify allows subpredicates to be labeled and outputs with its counterexample context the labels of those labeled subpredicates that somehow contributed to the counterexample context. For example, consider a Java program that contains the following assignment statement:

$$p.f = 10\,;$$

Ordinarily, this would be translated into an intermediate-language command like

$$\textbf{assert}\ p \neq \textbf{null}\ ;\ \ p.f = 10$$

which in turn would give rise to a verification condition of the form

$$\ldots\ \Rightarrow\ \ldots p \neq \textbf{null} \wedge\ \ldots$$

To use the labeling mechanism, the subpredicate $p \neq \textbf{null}$ in this verification condition is instead written as the logically equivalent

$$(\textbf{label}\ L\!:\ \ p \neq \textbf{null})$$

where L is a fresh label that encodes which program check the subpredicate represents. If the theorem prover outputs the label L with a counterexample context, the checker will report a warning of a possible null dereference in the Java assignment statement above.

We have found that the error kind, source location, and execution trace usually suffice to diagnose a warning produced by the checker. Therefore, the default in ESC/Java is to hide the theorem prover's full counterexample context from the user. However, there are times when the rest of the counterexample context does contain useful information. Though we have tried, we have not succeeded in finding a general scheme for automatically extracting all interesting parts of a counterexample context. Instead, we have focused on detecting a couple of common situations that without further information can be quite confusing to users [28].

1.4 Grappling with annotations

The last of the five big challenges regards program annotation. Is program annotation a task that programmers can reasonably be expected to perform? Are annotations understandable? Is the task of adding annotations too big a burden?

For both of the extended static checkers and their (different) annotation languages, we have found that the annotations describe programmer design decisions. That is, the annotations give properties that are relevant to the program's correctness, not obscure hints to the checking machinery that enable it to grind through the analysis of the given program. For example, an annotation may describe the decision that a particular method parameter should never be passed in as null or that the value of a particular integer field should always lie between 0 and the size of some array. This is good, because it means that programmers can understand what they write down as annotations. Moreover, the annotations serve as useful program documentation. And, unlike documentation written in a natural language, which can get out of sync with the program text, annotations can be mechanically checked to agree with the program text.

There are choices in the creation of a program checker's annotation language, because there are several different kinds of annotations that can be used to describe the same programmer design decisions. An important example of this choice is found in how the annotation languages of the two extended static checkers support writing down how data structures are represented. In ESC/Modula-3, the annotation language included abstract variables (fictitious variables whose values are given as functions of program variables) and abstraction dependencies (declarations that specify which program variables may be used in the representation of which abstract variables) [21, 18]. An annotation idiom commonly used in ESC/Modula-3 is to introduce an abstract field called *valid* in each object type, with the meaning that an object is valid if it has been properly initialized and its fields are in a consistent state [21, 9]. The field *valid* is then used as an explicit precondition of every routine that operates on such objects. To reduce the annotation burden involved in writing these pre- and postconditions, ESC/Java dispensed with abstract variables and abstraction dependencies in favor of object invariants, which declare what it means for the fields of an object to be in a consistent state, and which are then automatically used as preconditions of methods. The design choice to use object invariants led to further design choices, for example choosing when exactly an object invariant is supposed to hold. With its present object-invariant design, ESC/Java gives up on soundness in several ways where the abstract-variable design in ESC/Modula-3 did not have to. But the overall effect seems to be that the ESC/Java annotation language is easier to use. Trade-offs like this are essential to making more usable program checkers.

A critically important feature of the annotation language of an extended static checker is an escape hatch that suppresses the static checking performed, to be used when the checker issues spurious warnings that otherwise would be difficult or impossible to eliminate. Such escape hatches exist in traditional program checkers as well. For example, both Modula-3 and Java include type-cast expressions, which circumvent the strictness of the static type checker. To ensure the type safety of the running program, such type casts in Java and in the safe subset of Modula-3 are checked at run-time. Since ESC/Modula-3 and ESC/Java don't introduce run-time checks, their escape hatches belong under the rubric of unsound features.

For example, consider the following Java program fragment:

$$\begin{array}{l} y = \ x * x + 2 * x + 1\,; \\ z = \ \mathbf{new\ int}[y]\,; \end{array}$$

Because of properties of integers, y is always assigned a non-negative value (ignoring issues of arithmetic overflow), so the subsequent array allocation will never result in a negative-size array allocation error. However, an extended static checker is not likely to be equipped with the appropriate integer properties to deduce this fact automatically, so it will spuriously issue a warning. This warning can be suppressed in ESC/Java by adding a **nowarn** annotation on the line of the allocation:

$$\begin{array}{l} y = \ x * x + 2 * x + 1\,; \\ z = \ \mathbf{new\ int}[y]\,; \quad //@\ \mathbf{nowarn} \end{array}$$

or by instructing the checker to blindly assume the condition $0 \leq y$ after the assignment to y:

$$\begin{array}{l} y = \ x * x + 2 * x + 1\,; \\ //@\ \mathbf{assume}\ 0 \leq y\,; \\ z = \ \mathbf{new\ int}[y]\,; \end{array}$$

In this example, the escape hatch is needed because of the checker's limited support for non-linear arithmetic. In other situations, an escape hatch may be used as an alternative to writing down some complex program invariant. When an escape hatch is used, the user takes responsibility for those execution paths that are not checked by the checker.

We have found that annotating a program increases its number of source lines by about 10 percent. For programming teams that are serious about building quality into their software, this number does not seem excessively high. Programmers on such teams are already accustomed to writing similar annotations in natural language comments. But the start-up cost is still too high. For programming teams with large amounts of already written code, the initial investment of adding annotations to the legacy code seems daunting. The argument that the checker performs modular checking, which means one can annotate a class at a time, does not make the situation compelling enough, judging from our limited experience with programmers outside our projects. Even for programming teams that are just starting a new project, there's still the initial training cost before the team acquires at least one expert at the new checker. The fact that the technology is still new means the risk of using an extended static checker is higher than if its use were common.

Like type declarations, extended static checking annotations impose stronger invariants on the program. Decades ago, when the benefits of static type checking weren't generally accepted, the type checking community faced problems similar to the ones I've outlined above. But with the continued design and use of statically typed programming languages, the fact that type declarations must be given explicitly no longer poses a barrier to entry for such languages. (When was the last time you heard, "I don't want to use Java, because the burden of writing explicit type declarations is just too large"?) This piece of history gives hope to extended static checking technology, but more research is called for.

2 Future challenges

In the last ten years, the two extended static checking projects saw many research challenges, the first tier of which were overcome. More research challenges remain. In this section, I mention four of these and refer to some related work.

2.0 Reduce annotation burden

Although annotations capture programmer design decisions and provide a stylized way to record these, the reluctance to cope with the burden of annotating programs remains the major obstacle in the adoption of extended static checking technology into practice. Are there ways to use this more powerful checking technology at a reduced annotation cost?

One way to reduce the annotation burden is to not insist on modular checking. By spanning routine and module boundaries, the checker will need fewer annotations. Even if such a checker comes at a price of increased demand for computational resources (time, memory, disk space) or more complicated analysis techniques, it may reduce the overall effort of using the checker. Recently, an annotation-less static program checker called PREfix [34] has achieved good success in this area. Abstract interpretation [6, 5] is another program analysis technique that can find errors in whole programs without requiring annotation.

Another way to reduce the annotation burden is to develop *annotation assistants*, which infer annotations automatically from the program text. A tool called Daikon [11], which uses a mix of static and dynamic analysis techniques, infers likely invariants of a given program. An annotation assistant for ESC/Java, called Houdini [14, 13], is under development at Compaq SRC.

2.1 Understand sound modular checking

The input to a program checker includes not just the routine or class implementations to be checked, but also the declarations given in the *scope* of such routines and classes. For example, if the implementation of a routine r calls another routine p, then conventional programming-language rules for resolving names ensure that the declaration of p is in the scope of r's implementation. On the other hand, other procedure declarations, and the implementation of p, may not be in the scope of r's implementation. Modular checking can be performed so long as the input to the checker includes the scopes of the routines or classes to be checked. But is this modular checking meaningful? Is a verification condition produced in a limited scope somehow related to the verification condition that would have been generated if the whole program were in scope?

Modular checking is *sound* with respect to a verification-condition generator, if it doesn't miss any errors that the same verification-condition generator would have detected if given the whole program [18, 21]. For example, consider a limited scope M that's part of a program P, and consider an implementation r in M. (In Modula-3, the limited scope M corresponds to a module closed under imports, and P corresponds to all of the modules in the program.) Then, a theorem of *sound modular checking* takes the form

$$M \subseteq P \wedge \mathit{WellFormed}(M, P) \wedge r \in M \wedge \mathit{Pass}(r, M) \Rightarrow \mathit{Pass}(r, P)$$

where for any (limited or whole-program) scope X, the predicate $\mathit{Pass}(r, X)$ means that the checker issues no complaints about r when the checking is performed in the context of X. Soundness of modular checking is non-trivial and doesn't hold unless one restricts the programs under consideration. The predicate $\mathit{WellFormed}(M, P)$ says that module M and program P obey such restrictions.

In ESC/Modula-3, we tried hard to achieve sound modular checking. In contrast, we deliberately gave up on this chivalrous goal in ESC/Java, because it was not clear that the increased coverage that sound modular checking provides justifies the extra effort that it entails. But even if the extra coverage doesn't justify the extra effort, it may be enlightening to understand what sound modular checking really involves. As it stands, the soundness of modular checking for modern programming languages is an open problem [21].

2.2 Investigate more-than-types systems

Static type systems have had considerable success in popular programming languages. Consequently, the type checker helps enforce program invariants like "this variable is **null** or contains the address of a data record of type T". Experience has shown that the invariants imposed by a type system are easy to teach to programmers, helpful in finding common errors, and flexible enough for large classes of programs that programmers actually want to write. The flexibility comes in part from the fact that certain checks are not performed statically, but are instead enforced by simple dynamic checks. What are some stronger invariants that a programming language can reasonably enforce by a combination of static and dynamic checks? Can we do better than traditional type systems?

Extended static checking is a technique for finding errors in a program, not for providing guarantees about the program. Nonetheless, as a gedanken experiment, let us consider the possibility of making up for the (deliberate) unsoundness in ESC/Java's static checking by prescribing dynamic checks. Recall from Section 1.0 that by not checking for one kind of error, a checker may miss errors of others kinds, too. Thus, to guarantee any program invariant at all, we must examine every kind of unsoundness in the static checking.

One unsoundness in ESC/Java stems from **assume** annotations. The annotation statement **assume** p inhibits, from that program point onwards, ESC/Java's static checking for those program executions that reach the **assume** statement when p does not hold. An obvious way to make sure the inhibited checking is immaterial is to dynamically check p at the point of the annotation (reporting an error and halting the program if p does not hold). To keep the dynamic check simple may require restricting the annotation language, for example forbidding universal and existential quantifications.

Another unsoundness stems from the fact that ESC/Java does not enforce **modifies** clauses. For proper checking of the caller of a routine, one needs to know what parts of the program state the callee may modify. This is specified in **modifies** clauses, which require some form of abstraction in the annotation language [20, 21]. Lacking such support for abstraction (as in ESC/Java), one can turn to some form of approximation, but this gets tricky. Overestimating what the callee modifies results in spurious warnings in the caller; overestimating what the caller assumes to go unchanged by the call results in spurious warnings in the callee. To reduce spurious warnings, ESC/Java uses declared **modifies** clauses when reasoning about calls, but omits the corresponding checking for the callee. Using dynamic checking to make up for the lack of static **modifies** checking is difficult. Naively, it would require taking a snapshot of the entire program state on entry to a routine and then comparing the snapshot with the program state on exit from the routine. To implement more efficient checks may require enforcing a stricter programming methodology or restricting the expressiveness of the programming language.

A third unsoundness in ESC/Java stems from the relaxed rules about where object invariants are checked to hold. On entry to a call, ESC/Java assumes that all object invariants hold for all objects. But at call sites, ESC/Java checks the object invariants only for the actual parameters of the call. Adding dynamic checks to restore soundness would involve analyzing the program to determine which variables and object fields the callee may read or write. If this analysis is to be precise enough, one may need to rely on additional annotations or programming methodologies that restrict which objects may be reached from where. Several methodologies have been proposed to restrict "aliasing" of objects (see, *e.g.*, [3, 32, 38, 16, 1, 29]), but it seems there is still no practical and checkable solution to the part of this problem known as "rep exposure" or "abstract aliasing" [8].

Clearly, judging even from just the three kinds of unsoundness above, there are several hard problems to be solved before dynamic checking could complement extended static checking to achieve a sound system. Trying to increase the strength of guaranteed program invariants by instead adding static checks to systems that already prescribe dynamic checks can lead to similar problems. Many programming systems can dynamically check user-supplied assertions (see, for example, the pioneering work by Satterthwaite [35]). The object-oriented programming language Eiffel [27] and the Ada annotation language Anna [26] provide facilities for systematically introducing dynamic checks of assertions like preconditions and object invariants, but these languages have not been designed for the purpose of supporting complementary static checking. In fact, neither language includes **modifies** clauses, and in neither language is it possible to infer, from the specifications visible to a caller, what conditions the caller is responsible for establishing prior to a call (*cf.* [18]). So is there any hope of increasing the strength of program invariants that programming-language designs can incorporate and enforce?

A simple measure for increasing the strength of program invariants is to augment object-oriented type systems with *may-be-null types*. A may-be-null type is a variant record representing either the special value **null** or a value of some (non-null) object type. A deference expression $E.f$, where f is an object field, is then defined only when the static type of the expression E is a non-null object type. An expression of a may-be-null type can be cast to the corresponding non-null type; the cast fails (dynamically) if the expression evaluates to **null**. May-be-null types were used in CLU [25], but they seem mostly to have been forgotten since.

Another possible measure for increasing the strength of program invariants enforced by a programming language is to augment the type system with *dependent types* (see, *e.g.*, [2, 39]). These are essentially record, map, array, or object types with additional invariants. There is a temptation to stay within the realm of invariants whose checking is decidable, but I say why not leap above the decidability ceiling: by suitably restricting the annotation language, if an invariant cannot be checked statically, one can either fall back on dynamic checking or rely on the programmer to supply an **assume** statement (which may also require dynamic checking).

There are still problems with dependent types. One problem is the question of when to enforce the invariants. This problem is reminiscent of the problem of when to enforce object invariants, described above. Maybe one can restrict operations on values of dependent types in such a way that it becomes clear when the invariants should be enforced. Another problem with dependent types is the need for **modifies** clauses. To reason about particular program variables, one generally needs to know which variables may be modified by a call, but this leads to the problems with enforcing **modifies** clauses described above. It is possible that one could restrict the regions of a program where variables of dependent types are updated, in such a way that one does not rely on the exact values of these variables on entry to the regions, thereby possibly avoiding the need for **modifies** clauses. Another possible way out is to restrict one's attention to a functional language (in which variables are never changed), an approach explored by Augustsson in Cayenne [2] and by Xi and Pfenning [39].

In summary, the gap between the program invariants enforced by traditional type systems and the program invariants that extended static checkers check a program against seems to be wide. In response, I suggest investigating "more-than-types systems" which aspire to guarantee stronger program invariants than those guaranteed by traditional type systems, and which may rely on a combination of static checking above the decidability ceiling and dynamic checking. The research challenge is to investigate this space of programming-language designs to determine if there are more-than-types systems that are substantially more useful than traditional type systems. (In this volume, Schneider, Morrisett, and Harper discuss the combination of various static and dynamic checking techniques to enforce stronger security policies [36].)

2.3 Teach

One of the barriers to entry for extended static checking is that a regrettably large number of programmers don't really understand preconditions and invariants. When these concepts are taught in computer science curriculums, students tend to practice them only on paper. There's a large difference between turning in a homework assignment that is returned graded a week later and getting instant feedback from a mechanical checker. It seems to me that the current state of the art in extended static checkers, although designed to support programming in the large, would be quite instructional to use along with the compiler and type checker in early (and more advanced) programming classes. Even if the students wouldn't continue using an extended static checker outside the classes, the experience of using one with their programming assignments may teach them to think in terms of preconditions and invariants, which is likely to breed a new, better generation of programmers.

3 Conclusions

After a decade of research in the area of extended static checking, the first tier of research challenges has been overcome. Extended static checking seems promising as a technique to improve the quality of programs produced while programming in the large. However, challenges remain before extended static checking technology will be used routinely in program development in practice. As with the adoption of anything new, there are political and cultural barriers to break through. But I think there are also technical research challenges in getting the technology adopted. By investigating more-than-types systems, we may design programming languages that enforce stronger program invariants. By working on reducing the annotation burden and otherwise improving the user experience in applying extended static checking, we may produce better program checkers. By teaching a new generation of computer science students, we may raise better programmers. We can then hope for a future in which computer programs are more reliable.

Acknowledgments

ESC/Modula-3 was developed by Dave Detlefs, Greg Nelson, Jim Saxe, and the author, from the first half of the 1990's until 1996. ESC/Java was developed by Cormac Flanagan, Mark Lillibridge, Greg Nelson, Jim Saxe, Raymie Stata, and the author, starting in 1997. Damien Doligez, George Necula, Rajeev Joshi, Todd Millstein, and Silvija Seres worked on or with the extended static checking projects as SRC research interns. The projects also benefited from the help of other colleagues at Compaq SRC, including Rajeev Joshi, Steve Glassman, Allan Heydon, Marc Najork, and Caroline Tice. Rajeev Joshi, Greg Nelson, Jim Saxe, and Reinhard Wilhelm provided helpful comments on drafts of this paper.

References

0. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
1. Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97—Object-oriented Programming: 11th European Conference*, volume 1241 of *Lecture Notes in Computer Science*, pages 32–59. Springer, June 1997.
2. Lennart Augustsson. Cayenne — a language with dependent types. In *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34, number 1 in *SIGPLAN Notices*, pages 239–250. ACM, January 1999.
3. John Boyland. Alias burying: Unique variables without destructive reads. *Software—Practice & Experience*. To appear.
4. Edmund Clark. Language constructs for which it is impossible to obtain good Hoare-like axioms. *Journal of the ACM*, 26(1):129–147, January 1979.
5. Patrick Cousot. Progress on abstract interpretation based formal methods and future challenges. In *Informatics—10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.
6. 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, January 1977.
7. Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 84–96, January 1978.
8. 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.
9. 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.
10. Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.
11. Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *ICSE 2000, Proceedings of the 22nd International Conference on on Software Engineering*, pages 449–458, 2000.

12. Extended Static Checking for Java home page, Compaq Systems Research Center. On the web at `http://research.compaq.com/SRC/esc/`.
13. Cormac Flanagan, Rajeev Joshi, and K. Rustan M. Leino. Annotation inference for modular checkers. *Information Processing Letters*. To appear.
14. Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. Technical Note 2000-003, Compaq Systems Research Center, 2000.
15. Steven M. German. Automating proofs of the absence of common runtime errors. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 105–118, 1978.
16. John Hogg. Islands: Aliasing protection in object-oriented languages. In Andreas Paepcke, editor, *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'91)*, pages 271–285. ACM Press, October 1991.
17. S. C. Johnson. Lint, a C program checker. Computer Science Technical Report 65, Bell Laboratories, Murray Hill, NJ 07974, 1978.
18. K. Rustan M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995. Technical Report Caltech-CS-TR-95-03.
19. K. Rustan M. Leino. Ecstatic: An object-oriented programming language with an axiomatic semantics. In *The Fourth International Workshop on Foundations of Object-Oriented Languages*, January 1997. Proceedings available from `http://www.cs.williams.edu/~kim/FOOL/`.
20. 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.
21. K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. Research Report 160, Compaq Systems Research Center, 2000.
22. K. Rustan M. Leino, Greg Nelson, and James B. Saxe. ESC/Java user's manual. Technical Note 2000-002, Compaq Systems Research Center, October 2000.
23. K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. In Bart Jacobs, Gary T. Leavens, Peter Müller, and Arnd Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*, Technical Report 251. Fernuniversität Hagen, May 1999. Also available as Technical Note 1999-002, Compaq Systems Research Center.
24. K. Rustan M. Leino and Raymie Stata. Checking object invariants. Technical Note 1997-007, Digital Equipment Corporation Systems Research Center, January 1997.
25. Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Electrical Engineering and Computer Science Series. MIT Press, 1986.
26. David C. Luckham. *Programming with Specifications: An Introduction to ANNA, a Language for Specifying Ada Programs*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.
27. Bertrand Meyer. *Object-oriented Software Construction*. Series in Computer Science. Prentice-Hall International, New York, 1988.
28. Todd Millstein. Toward more informative ESC/Java warning messages. In James Mason, editor, *Selected 1999 SRC Summer Intern Reports*, Technical Note 1999-003. Compaq Systems Research Center, 1999.
29. Naftaly H. Minsky. Towards alias-free pointers. In Pierre Cointe, editor, *ECOOP'96—Object-Oriented Programming: 10th European Conference*, volume 1098 of *Lecture Notes in Computer Science*, pages 189–209. Springer, July 1996.

30. Greg Nelson. Combining satisfiability procedures by equality-sharing. In W. W. Bledsoe and D. W. Loveland, editors, *Automated Theorem Proving: After 25 Years*, volume 29 of *Contemporary Mathematics*, pages 201–211. American Mathematical Society, 1984.
31. Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.
32. James Noble, Jan Vitek, and John Potter. Flexible alias protection. In Eric Jul, editor, *ECOOP'98—Object-oriented Programming: 12th European Conference*, volume 1445 of *Lecture Notes in Computer Science*, pages 158–185. Springer, July 1998.
33. D. L. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, May 1972.
34. PREfix. Intrinsa, Mountain View, CA, 1999.
35. E. Satterthwaite. Debugging tools for high level languages. *Software—Practice & Experience*, 2(3):197–217, July–September 1972.
36. Fred B. Schneider, Greg Morrisett, and Robert Harper. A language-based approach to security. In *Informatics—10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.
37. Richard L. Sites. *Proving that Computer Programs Terminate Cleanly.* PhD thesis, Stanford University, Stanford, CA 94305, May 1974. Technical Report STAN-CS-74-418.
38. Mark Utting. Reasoning about aliasing. In *Proceedings of the Fourth Australasian Refinement Workshop (ARW-95)*, pages 195–211. School of Computer Science and Engineering, The University of New South Wales, April 1995.
39. Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Conference Record of POPL'99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 214–227, January 1999.