Bart Niswonger
CSE 503: Winter 2002
Tool Evaluation: ESC/Java

Introduction

This paper discusses the Extended Static Checker for Java (ESC/Java), a static Java program verifier from Compaq SRC[1]. This tool is static in that, like a compiler, it examines the source code without any runtime information. As a verifier it is both unsound – it may report an error which does not exist – and incomplete – it may not report an error which does exist – as a result, the authors refer to it as a checker rather than a verifier. While these two properties seem undesirable, ensuring both of them would imply that the tool would not terminate in at least some cases. The hope is that false positives and false negatives occur rarely enough that the tool is helpful for a user, while ensuring efficiency and termination, also useful for a user.

The problem

The aim of program verification is to discover errors in a program. Starting from some specification, the idea is to prove at each program point that the specification is not violated. Specifications can be expressed in any number of ways from Hoare triples and algebraic languages to model-based languages like Z. In its full glory, automatic program verification is undecidable: verifying whether a program terminates is the halting problem.

The authors of ESC/Java argue that while full automatic verification would be nice if it were achievable, an automatic tool which catches many common errors is possible, and therefore more useful. The idea of an automatic tool is important to them. Program verification tools have been developed, but the undecidability of the problem implies they require a user. When dealing with large bodies of code, the job this user faces may not be possible. Automatic tools, the ESC/Java authors argue, are crucial in this context. Their approach is to identify some set of common errors which can be detected automatically, and to explicitly ignore others: ESC/Java is unsound and incomplete by design, and the authors attempt to be very clear exactly what types of errors are missed and they provide techniques to ignore false positives. The contribution of this very pragmatic approach is to demonstrate that in fact many errors which are difficult for programmers to discover can be automatically checked.

In the following section I will discuss my interest in this tool, and my approach to evaluating it, including a description of the source code I used. Following that, I will describe the tool itself, and lay out my understanding of the authors goals. In the final section, I will discuss how well the tool achieves those goals based on my experience and conclude my evaluation.

[1] ESC/Java is a descendent of the Extended Static Checker for Modula-3. In fact, the authors claim that the underlying ideas can be applied to any language in which address arithmetic cannot be performed.

The evaluation

Motivation

I find this tool interesting for a few reasons. First, I have always been a fan of using the compiler as an aid in code development – it lets me not worry about syntax and I can use it to catch a class of errors. ESC/Java feels like, by design it turns out, a more powerful type checker meaning that now I have even less to worry about. Second, when I write code there are things I would like to express which are not expressible cleanly in Java, and are only caught at runtime. An example might be that one method should be called before any other method in the class is called. It is possible to make sure this happens in Java: there needs to be some variable that the first sets and all other methods make sure it is set. While this solution works, I argue that it is both clumsy, and likely that additional plain text comments would be needed to explain what is happening, i.e. it is unclear. ESC/Java provides another way to do this, which is checked at compile time, and which is expressed in a more natural language. Third, I am not sure I believe in specifications as a tool for programmers. The problem of the source and the specification getting out of sync is serious. I am interested in examining to what extent this tool prevents that problem.

Methodology

In order to explore ESC/Java, I decided to apply it to some code I have been developing as part of my quals project. I will describe the code in a moment, but first there is the more general question of how to evaluate this tool.

A simple metric is to compare the total number of lines of code before and after annotating it. This would seem to suggest the amount of effort a user must go through to use the tool. While this might be interesting from an HCI perspective, it does not seem to cleanly address whether the concept underlying this tool 1) satisfies the goals of the authors, or 2) is useful in practice. I find these two questions more compelling when examining ESC/Java, as it is a research prototype and therefore may not have the most polished user experience. The first question, how well does ESC/Java satisfy the goals of its authors, seems relatively clear cut. The second, whether the tool is useful, could be rephrased as: what benefit does the user gain by using the tool? One metric might be the number of bugs found, but in fairly well tested code it seems like this number might be rather misleading. For me, the attraction of this tool is not in applying it to some static code base, but for use when developing the code to simplify the task of chasing down why assumptions fail while developing. Unfortunately, due to time constraints, I have not been able to pursue this avenue. Another, more qualitative, approach to analyzing the benefit is to explore what kinds of things a developer might want to express, and how easy it is to express them using tool. This becomes a critique in part of the specification language rather than the concept, however to the extent that the specification language expresses the concept, this seems reasonable.

In my analysis, I have focused both on the degree to which ESC/Java implements the goals of its authors, as far as I could determine them, and also the degree to which it facilitated the expression of the ideas I wished to express. These questions are orthogonal to that of the specification and the source code getting out of sync, which I also address.

Test code

The program I tested it on implements a simulation of a simple peer to peer search tree algorithm. It consists of roughly 2400 lines of Java code, including comments, in 16 files with 4 interfaces and 14 classes. During execution there are two threads, one for the GUI and another running the simulation. It makes extensive use of the Java Collection hierarchy, and the Swing GUI API.

A few aspects of this program are worthy of highlighting:

- It uses APIs for which the source may not be available. Many tools require the entire program source to be available, however ESC/Java does not, as we shall see. This program allows us to explore how well ESC/Java supports this situation.
- It includes interfaces and several (small) class hierarchies. In object-oriented programming, inheritance is common. What should a specification language do: should it follow the inheritance hierarchy? ESC/Java argues that it should. What needs to be verified about a class hierarchy? What about multiple inheritance?
- It is multithreaded. While ESC/Java does support verification of synchronization techniques, I did not explore this area.

The tool

The basic user experience with ESC/Java is to run the tool, as one would a compiler, and read the (rather copious) set of warnings which are output. These warnings tell the user lines in the source code which may cause particular runtime errors. The authors have chosen a set of runtime errors including: illegal cast, null dereference, negative or out of bounds array index, divide by zero, deadlock and race condition. This set is not complete, but it was chosen to include common errors, and to include errors that were possible to detect efficiently. This set is considered larger than was predicted to be possible, however I saw no arguments as to whether more could be done, or how much more.

Once the user has a set of warnings, they try to get rid of them, just as they would compiler errors. There are two basic approaches: change the actual source code or add annotations to it. It is up to the user to decide which makes more sense in a given situation. In my experience, where the program already worked well, it was relatively rare that I changed the source, primarily I added annotations. Annotations are embedded in comments so that the java compiler will ignore them. They are expressed in the ESC/Java specification language, which is quite similar in syntax to Java itself, and which I will describe briefly below. The annotations act to guide the tool: some place additional constraints on what a legal program is, others tell the checker to ignore certain types of warnings (possibly introducing more unsoundness, but avoiding some of the incompleteness in the system). After the user adds annotations, they rerun the tool, frequently getting new warnings. The process is repeated, until finally they have a clean run.

ESC/Java works by taking the source code annotations and translating them into verification conditions. It does a modular inter-procedural analysis: it analyses the body of a procedure only once and uses pre- and post-conditions to analysis call sites. The translation to verification conditions is done using Dijkstra's weakest precondition transformations. The verification conditions are feed into an automatic theorem prover which has the task of deciding whether the conditions are satisfied. It does this by attempting to prove the negation: failure to satisfy the negated condition implies the condition is valid and the program meets the specification. Interestingly, the theorem prover is sound, it is in the process of generating verification conditions that allowing for unsoundness (i.e. ignoring certain error types) is said to be useful [ESC/159: 32].

Specification Language

The specification language consists of pragmas, some of which take specification expressions. The pragmas are instructions to the tool (e.g. `assume` or `assert`) and come in four flavors: lexical, statement, declaration and modifier. There is only one lexical pragma, `nowarn`, which suppresses a particular warning type. Statement pragmas are analogous to Java statements and include `assert`, `assume`, `unreachable`, and `set`. When ESC/Java encounters an `assert`, it will issue a warning if it is unable to prove the expression true at that program point. `assume` is the flip side: ESC/Java will assume the expression to be true without verifying it at all. Clearly this is a source for unsoundness, as the tool trusts the user to be correct, however it is important that the user have a way to express information the tool is not sophisticated enough to discover. `unreachable` is used to indicate some line should never be reached, much like `assume false`, however a bit more expressive. `set` is how one changes the value of ghost variables, as described below.

Declaration pragmas are similar to Java declarations. There are three of them: `invariant`, `axiom`, and `ghost`. Declaring an `invariant` causes the tool to `assert` that expression at the beginning and end of each procedure. An `axiom` is similar, but causes ESC/Java to `assume` the expression at the start and end of each procedure. A `ghost` declaration is how one gets a ghost variable, described in more detail below.

The final class of pragmas are the modifiers. This set includes variable modifiers such as `non_null`, `spec_public`, `monitored_by` and procedure modifiers like `requires`, `modifies`, `ensures`. Variable modifiers adjust how the tool handles the modified variable. For example, a variable with a `non_null` modifier is checked at every assignment to ensure that the assignment has not made it `null`, and is assumed at every use to be non-`null`. `spec_public` variables are treated as public variables by ESC/Java (and therefore can be used in a `requires` pragma for a public method), although they are actually declared to be private to a Java class. The `monitored_by` modifier is used to indicate a variable that is protected by a lock. This is the method whereby the user can express synchronization assumptions and requirements.

Procedure modifiers are used to express pre- and post-conditions. `requires` represents a pre-condition, while `ensures` is a post-condition (and `exsures` is a post-condition if some exception `T` is thrown). `modifies` is loosely a post-condition. It is useful when some `non_null` instance variable is not initialized by the constructor, but by some method the constructor calls. If the callee is declared to modify the variable, the caller (the constructor) can assume that the callee has set it to be non-null. The callee should then be checked to be sure it actually does modify the variable and in the case of a `non_null` variable, sets it to a non-`null` value. Unfortunately, the current implementation of ESC/Java does not verify that variables declared to be modified by procedures are actually changed. There was no explanation for this oversight, although the sense was that it is possible, just more work than they felt was necessary at the time.

Most pragmas expect some specification expression. These expressions are very similar to Java expressions in syntax, name-resolution and type-checking rules, although they do differ in other ways. Specification expressions must not include sub-expressions with potential side effects, so methods, assignment, and object creation are all excluded. In addition, specification need to reason about some entities which are not available in Java, so extra syntax has been added. There are quite a few of these but this list should give a flavor: the type of a variable (`\type(x)`), subclass relations (`<:`), existential and universal quantification (`\forall` and `\exists`), implication (`E ==> F`) freshness (`\fresh(x)`) and value on entry (`\old(x)`). Expressions can be built using these, or not. For example, if `x` is a variable and `Chair` is a class, the expression `\typeof(x) == \type(Chair)` would evaluate to true only if `x` was of type `Chair`. As another example, `x != null` would be true only if `x` could be shown to be non-`null`.

Ghost variables

A ghost variable is just like any other variable in Java, except that it is only visible to ESC/Java[2]. The current implementation also forces them to be public, but that seems only an implementation

[2] The original ESC for Modula-3 had a much more sophisticated version of the ghost variable construction called abstract variables. Abstract variables allowed the side effects of a procedure to be described in terms of variables within the clients' scope.

detail. They can be used exactly like other variables are used, however their primary use is to express truths about the program which could be derived from instance state or runtime information, but is not possible to describe in ESC/Java. The most common use in my experience was to represent the type stored in a collection class. Every class which is a subclass of the Collection interface inherits a ghost variable `elementType` which it is expected will be set (by the client of the collection class) to the type of the elements in the collection. Accessor methods are then annotated to include a post condition saying that their result is of type `elementType`, and therefore casts can be verified.

These variables can also be used to solve the design problem I alluded to in the introduction: I have some method which should be called before any other in the class. One solution using plain Java is to use an instance variable which is set by the first method and checked by all others. This increases runtime overhead of those methods and seems to me likely to be expressed unclearly, i.e. it is likely some comments would be needed to say that this method requires some other method to be called first. Using ESC/Java I can use a ghost variable which is set in the first method and which is listed as a pre-condition for all other methods. In this approach the cost is paid upfront at compile time, and it could be argued that the pre-condition is a more accurate manner in which to express the situation. I find this example interesting because it starts to get at the issue of the separation of source from specification. Notice that when I add a new method to this class, in either case, I must remember to include the additional code or pre-condition, and in neither case will I be told if I forget to do so. So the maintenance cost of this code is roughly the same in either case.

Their goals

Now that I have given a sense for the context in which ESC/Java is being evaluated, and a feel for the tool itself, I will discuss the goals of the authors. There are two questions. First is whether they identified the correct set of goals, and second is how well they achieved them. The first is a question of validation, and the answer is going to be rather vague because it is a subjective question and it seems to me that they got the set of goals roughly right. The second is more about verification and therefore can be more objective, and again it seems to me they basically meet the mark.

This project builds on many years of work in the fields of static checking and program verification. However earlier work often made simplifying assumptions which resulted in implementations that were not useful in practice. The authors primary goal was to leverage this previous work, but to build a system which was useful. To be useful, they wanted it to be automatic, to produce meaningful warning messages, to work for “real” programs, to catch common errors, and yet to have a fairly simple annotation process[ESC/159:3].

Are these the right goals?

My feeling is that they are more or less the right goals. ESC/Java is a program verifier. Program verification would clearly be valuable for some tasks, if it were easier to perform. The authors have identified some of the failings of previous systems, and sought to rectify them in this, thereby ending up with a tool which is (somewhat) easy to use.

The goal of building a useful system certainly seems like a good one. The trick comes in the definition of useful. The five conditions listed above seem like a good start, but they do not seem complete. For instance, the basic usage pattern I described above is iterative, and in an iterative system prompt feedback is very useful. Yet, the issue of performance is not directly included in any of the five conditions. Given the state of the art in theorem provers, this is probably a good move. In fact, their argument for the system being automatic is largely to deal with the problem of performance.

If it doesn't require a user, it could be run overnight, however unfortunate that would be. An iterative system which runs overnight would not be very useful. Happily, the performance is pretty good, although they do lament the complete lack of predictability in several places.

Another aspect of a useful system, but which I never saw expressed as a goal of theirs was that the warnings be reasonably accurate, and that most errors (of the included types) were found, i.e. that the unsoundness and incompleteness be rather minimal. This seems important to me, perhaps they simply thought it was implicit. Unfortunately, it is very hard to determine how well they achieved this goal, I certainly do not have the sophistication in this area to create examples for which ESC/Java will fail. The pragmas exist to help the user tune the soundness and completeness, and they have caution messages which are sometimes issued when soundness is in question, but it is still rather like a black box which may sometimes lie.

The desire for a tool which works on "real" programs meant they needed to support several language features often ignored in earlier work. These include (in their words) concurrency, dynamically allocated data, object oriented programming, and the use of libraries (where source code may be unavailable) [ESC/159:2]. Again, this seems like a good set. Certainly for my test code, this was sufficient. Additional concerns might arise when dealing with distributed applications, systems which manage their own memory or have to respond to hardware interrupts. They do not mention any of these, but none seem to be insurmountable using this tool.

The last piece of this picture, which I feel they did not explicitly discuss, is the fact that specifications and source code evolve at different rates. A goal of any useful program verification system must, in my mind, address this issue. One way is to automatically derive the program from the specification, or visa versa. Another is to have them independent, but have tools to ensure they are in agreement. In putting both together in a single file, ESC/Java has gone a long way to helping them evolve at the same rate: it seems much more likely that a programmer will update the pre- and post-conditions in the specification when they are editing the source if both are written together. Additionally, this tool ensures that the program is at least as valid as the specification expressed in it. A further question might be whether the specification language allows the kind of expressiveness available through Z or some other specification language.

Do they achieve these goals?

Primarily they do achieve the goals they set out to. They also achieve several of the goals I would expect such a system to.

I believe they would argue that they achieve their goal of being automatic, and in some sense they are correct. However, they also expect the user to go through several iterations of adding annotations. This strikes me as having a user in the loop, guiding the verifier. In this way, I feel it is not really all that automatic. If performance were truly poor, their definition of automatic would be particularly questionable. On the other hand, I found it useful to be guided towards my errors, i.e. as with a compiler this tool gives the user a list of things to fix. One question that came to mind while using the tool was how my experience of annotating an existing program would differ from annotating a system as it was developed, working from a static specification. In that case, perhaps ESC/Java would feel much more automatic, given that the source code would be likely to change more than the annotations.

In order to be useful, a tool like this certainly must output helpful warnings. I found their warnings to be mostly helpful, although occasionally rather obtuse. They have done a fair amount of work in this area. I was originally very impressed by ESC/Java's ability to output suggested pragmas which would get rid of certain errors. These proved to be a useful way to learn the tool, at least to a point, but beyond that were too simple to be helpful. I was also interested by the fact most of their pragmas were only needed in order to give useful error messages, since in essence they could all be reduced to assert and assume. By far the worst warnings to deal with are those indicating an invariant has been violated. In this case the tool reports the invariant in question, the line on which it was shown to be violated, the counter examples which show how it was violated, and an execution path whereby it could have been violated. The amount of information is impressive, although the counter examples are very hard to read (the user manual does not even attempt to describe the syntax). However, even with all that, determining why it failed and how to fix it is not trivial.

Along the same lines, reasonable accuracy is crucial. If most of the warnings are false positives, no one will take the time to filter through them to find the few true errors. I found no truly false positives. By that I mean I never had to use nowarn or assume to suppress a warning that I could not get rid of in other ways. Some of the warnings I got did take rather a long time to figure out exactly how to suppress them 'correctly', so the temptation to use these rather heavy handed methods was strong. I was interested to read in [ESC/159] that they actual encourage users to use these pragmas rather than hunt down tricky ways to satisfy the theorem prover, although additional unsoundness is introduced. It was not clear to me why they recommended this.

One of the hardest goals for me to analyze was the desire to catch common errors. What errors are common? Even so, I feel like they did a good job here. They catch a few very simple errors: array bounds and null dereferencing, and then some very sophisticated ones: deadlock and race conditions. Looking at OS security weaknesses would seem to suggest that array bounds checking errors are quite common, even in very mature code. Security weaknesses are interesting because they are basically well publicized bugs in code which traditional development and testing has missed. From my own development experience, null pointers and synchronization errors are problematic: the first because they are common though usually easy to find, and the second because they can be truly hard to find. I was sorry to not fully explore their synchronization pragmas. Since I have no other way to find bugs in my test code, I have no idea if there are things lurking which this tool was unable to find.

Part of being useful is having a fairly simple annotation process that allows users to express what they want. I was happy using their language. It was very natural to write. It seemed like a good language to express restrictions (`x` should not be `null`), but not well suited to expressing 'freedoms' (`y` can be `null`). At several points when writing pre- and post-conditions I wished I could express these freedoms. They are not needed, obviously the tool will automatically find them, however I find it helpful to know, particularly when trying to determine the correct fix. If you know something was designed to occasionally return `null`, then you are better off than if you are not sure. I found myself writing specifications like `y == null || y != null` which does get the point across, but in a rather flimsy manner.

My test program was "real" by their definition in that it was object oriented, dynamically allocated data. had concurrency and used libraries. However, it was not distributed (it only simulated a distributed algorithm), did not manage its own memory, or handle interrupts. Also, I didn't not check for deadlock or race conditions, so I have little to say about their support for concurrency. Since most distributed systems can be modeled as state machines, it seems likely that this tool would be able to help to some degree. The place where it seems unlikely to succeed is in reasoning about ordering of messages, but then so are humans. A program which manages its own memory may allow forms of sharing that the specification language is not expressive enough to capture. Sharing without locks seems like the exception rather than the rule however. Interrupts I think could be expressed as exceptional exit conditions are now.

I was quite happy to find that subclasses inherited specifications from their super-classes. Subclasses can also extend the specifications of their ancestors, which causes problems since the analysis is done statically with no runtime type information. Multiple inheritance introduces additional unsoundness, for the same reason. Neither of these were an issue for me, as I did not have need to extend the base class specification. In fact, it seems rare that such extensions would be needed. Protection, on the other hand, is handled in a rather non-intuitive way. Variables in pre- and post-conditions must be available to the caller (unless modified with `spec_public`). While this makes some sense, it is strange when you are writing the conditions in the file containing the callee and frequently want to talk about how the internal state of the object changes. Their arguments are convincing however, that by having those variables in the pre- and post-conditions you are exposing them to the caller, forcing the caller to depend upon the internal representation of your class. From reading the description of abstract variables in [ESC/159] I get the sense that they would be much more natural to use, making this a non-issue.

Constructors are treated somewhat specially in that they are expected to initialize any non-`null` variables. They also set a special ghost variable `owner` which is used to track aliasing. The primary use of this variable appears, from my experience, to be ruling out counter examples that the theorem prover finds where two objects `a` and `b` both point to the same object `c`. If `a` changes `c`, then `b` may have an invariant violated. For instance, if `a` and `b` are instances of a `Set` class with an instance variable `size`, and `c` is an array of elements, then a reasonable invariant might be that `size` is equal to the number of elements in the array. If `a` adds something to the array, and updates its own `size` variable, then `b`'s `size` variable is incorrect. The `owner` variable is used to demonstrate that a particular object belongs to whatever object created it. So if `a` and `b` both allocate their own array, then one array will indicate `a` as the owner and the other will indicate `b`. The theorem prover can then be sure that the array will not be aliased.

Libraries are omnipresent in Java, so the ability to support them is essential. ESC/Java supports missing source code in three ways. First, it is a modular checker: it uses the procedure modifier pragmas of the callee when examining the caller without explicitly examining the callee. This means that only the specification of a procedure is needed when analyzing a callsite., Second, if no specification is found for a method (even if there is source), one is generated by examining the signature. It will not be very sophisticated, but it means that you do not have to annotate all your code before running the tool. Finally, it allows files which only include specifications. These specifications are assumed to be correct, as there is no way to verify them. ESC/Java comes with specification files for the core of Java, including the Collection hierarchy. This was very helpful. However it does not include specifications for the Swing API. As a result I found myself writing specifications for procedures for which I was unsure of the exact semantics. This is not easy. Happily with Java, you can get the source, but that defeats the purpose. I don't know how much better one could do than they have done, perhaps the tool could analyze the byte code to verify the specifications, but even then there are implications of the internal state which are unclear and difficult to infer when writing the specifications. It remains a hard problem.

As I mentioned above, one of my goals for a verification tool is that it not allow the source and the specification to diverge by much. ESC/Java seems to do a good job in this regard, whether this was a goal of its designers or not. The only way for them to diverge is for the tool to never be run. However this comes at a cost. It is much easier to let the specification diverge from what was intended, i.e. ESC/Java makes it easier to build a program correctly, but it may make it harder to build the correct program. There is a related project working developing tools around the Java Modeling Language (JML) which may help in this regard. The JML and ESC/Java's specification language are related, so tools which work on the JML should work for ESC/Java. A tool which extracted the specification from the annotated source code and presented it in a more traditional manner would go a long way towards preventing the specification from wandering. Of course one what went the other direction would also be very nice.

Conclusions

All told, I enjoyed using this tool and plan to continue. It allowed me to express things about my program which I could only do in comments before, and it convinced me that it really could find fairly subtle errors. The fact that the specification and the source code are tightly coupled was very important to me, as I do not want to manage both. It was not tremendously helpful in finding bugs in the code I tested it on, it basically only forced me to cleanup my act with respect to user input and null parameter checking. However it did find one place in which I had made a design error, expecting a set would never be empty when it could be. I knew about the error, and had hacked a fix for it, but using ESC/Java I was able to find a cleaner fix. I am very interested to see how useful it is during development, and I am looking forward to actually having a specification, knowing when I need to check for null and when I don't. It will also be interesting to learn when it makes mistakes. I was happy using it and found no glaring holes, although I did wish it was more tightly coupled with the compiler and had an emacs mode since the output is a bear to dig through.