US 20020112201A1

(19) **United States**

(12) **Patent Application Publication**

Flanagan et al.

(10) **Pub. No.: US 2002/0112201 A1**

(43) **Pub. Date: Aug. 15, 2002**

(54) **METHOD AND APPARATUS FOR AUTOMATICALLY INFERRING ANNOTATIONS FOR AN EXTENDED STATIC CHECKER**

(76) Inventors: **Cormac Andrias Flanagan**, San Francisco, CA (US); **K. Rustan M. Leino**, Sunnyvale, CA (US)

Correspondence Address:
Pennie & Edmonds, LLP
3300 Hillview Avenue
Palo Alto, CA 94304 (US)

(21) Appl. No.: **10/007,113**

(22) Filed: **Dec. 4, 2001**

Related U.S. Application Data

(60) Provisional application No. 60/251,304, filed on Dec. 4, 2000. Provisional application No. 60/251,305, filed on Dec. 4, 2000.

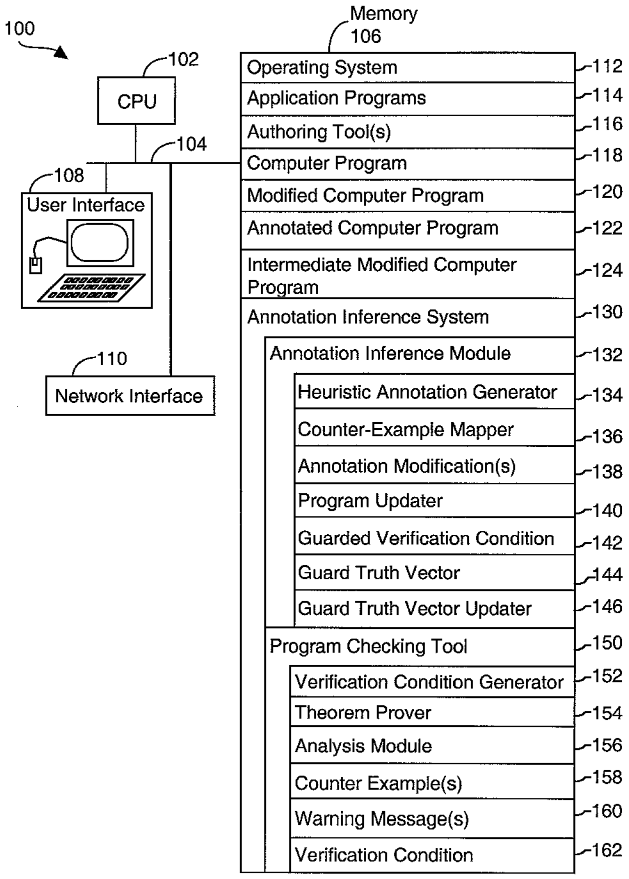
Publication Classification

(51) **Int. Cl.**^{7} **H04B 1/74**

(52) **U.S. Cl.** **714/42**; 714/38

(57) **ABSTRACT**

A system, method and computer program product for annotating a computer program. The method includes inserting a set of heuristically derived candidate annotations into the computer program and converting the computer program into a verification condition-which includes a set of guards corresponding to the set of candidate annotations. Initial truth values are assigned to the guards. A theorem prover is applied to the verification condition, and the counter-examples are mapped into one or more annotation modifications. The truth value of at least one of the guards corresponding to the one or more annotation modifications is updated. The theorem proving, mapping and truth value updating steps are repeated until the theorem prover produces no counter-examples that are suitable for mapping into an annotation modification. The resulting annotation modifications are applied to the computer program. The system and computer program product implement this method of annotating a computer program.



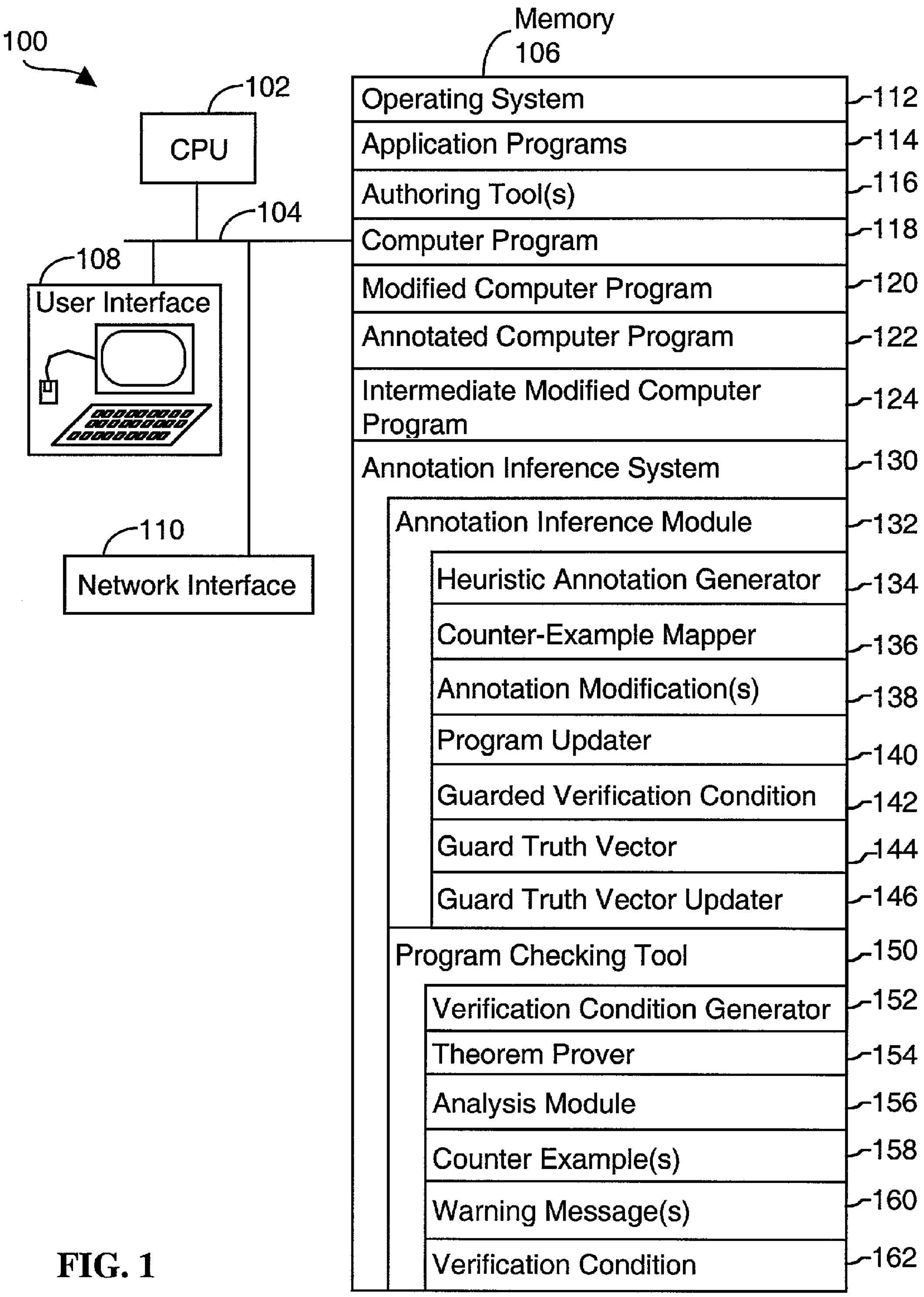

100
102
CPU
104
108
User Interface
110
Network Interface
Memory
106
Operating System
112
Application Programs
114
Authoring Tool(s)
116
Computer Program
118
Modified Computer Program
120
Annotated Computer Program
122
Intermediate Modified Computer Program
124
Annotation Inference System
130
Annotation Inference Module
132
Heuristic Annotation Generator
134
Counter-Example Mapper
136
Annotation Modification(s)
138
Program Updater
140
Guarded Verification Condition
142
Guard Truth Vector
144
Guard Truth Vector Updater
146
Program Checking Tool
150
Verification Condition Generator
152
Theorem Prover
154
Analysis Module
156
Counter Example(s)
158
Warning Message(s)
160
Verification Condition
162

FIG. 1

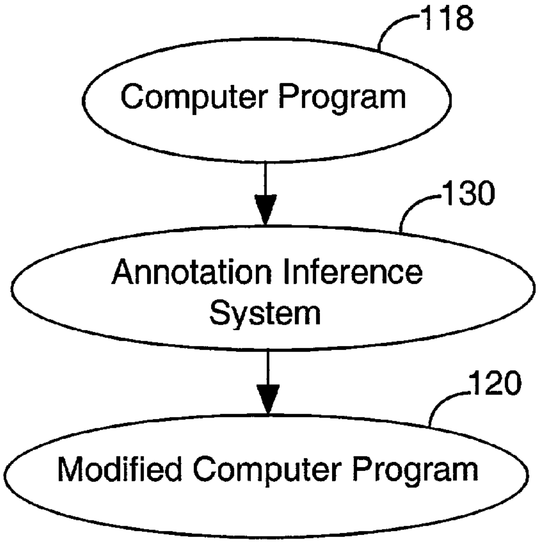


FIG. 2

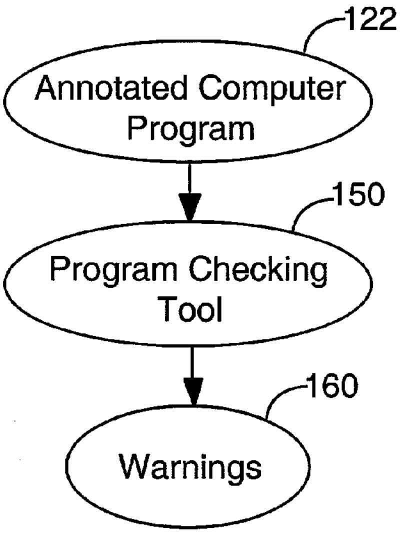


FIG. 3A

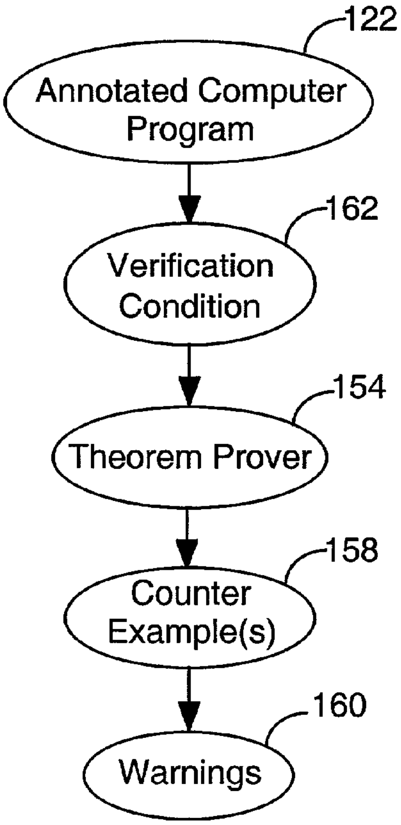


FIG. 3B

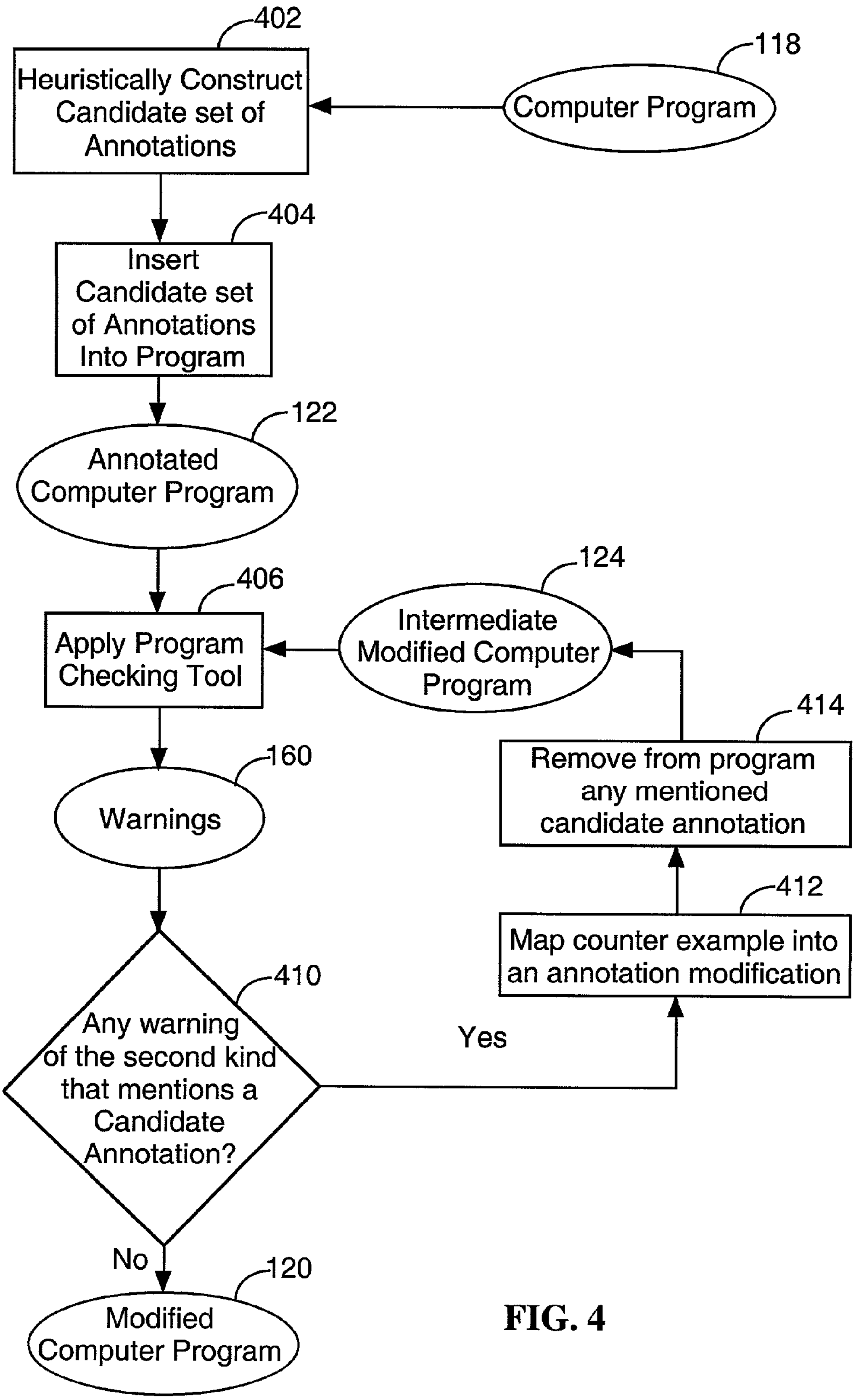

402
Heuristically Construct Candidate set of Annotations
118
Computer Program
404
Insert Candidate set of Annotations Into Program
122
Annotated Computer Program
124
Intermediate Modified Computer Program
406
Apply Program Checking Tool
414
Remove from program any mentioned candidate annotation
160
Warnings
412
Map counter example into an annotation modification
410
Any warning of the second kind that mentions a Candidate Annotation?
Yes
No
120
Modified Computer Program

FIG. 4

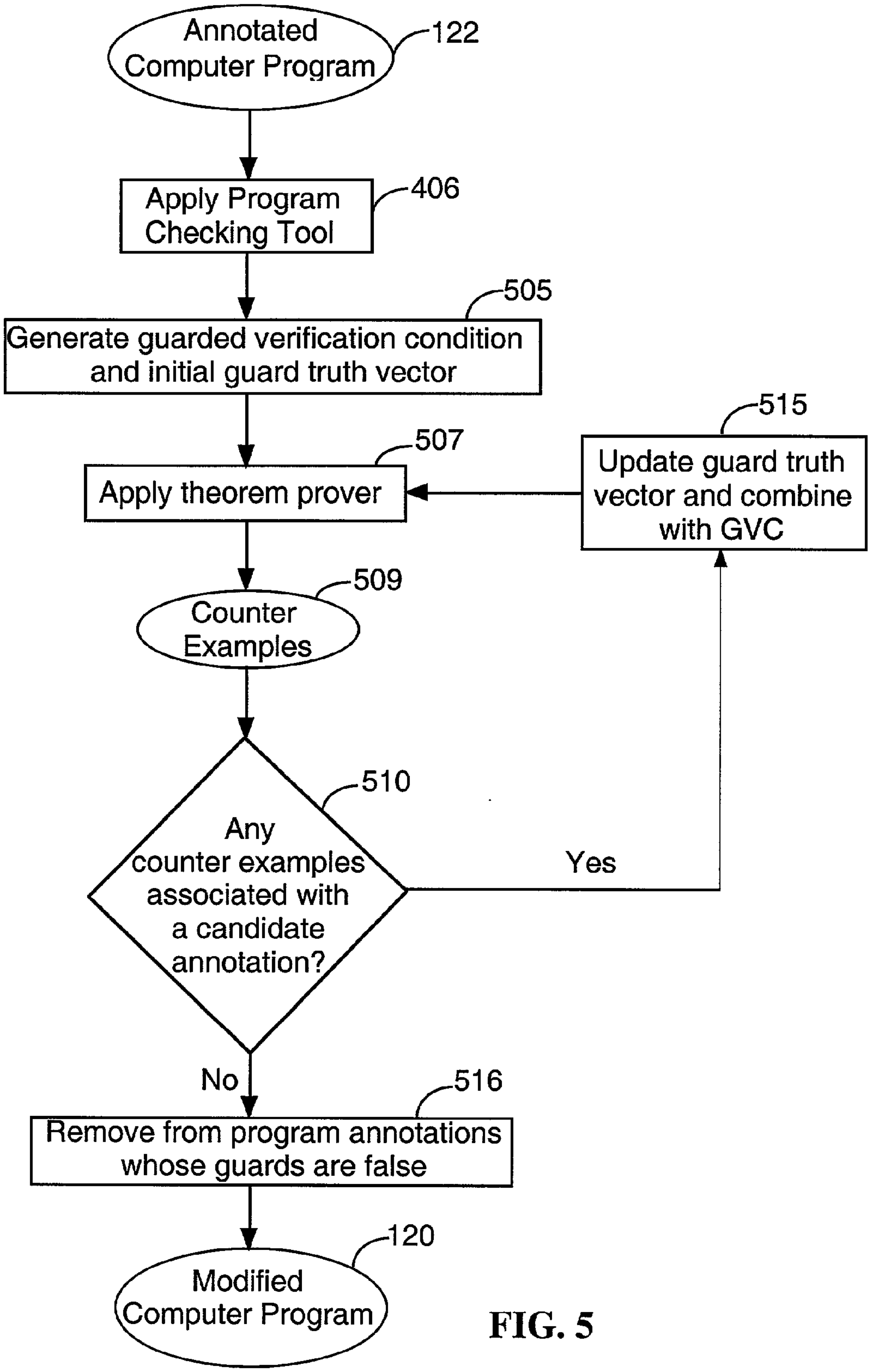

Annotated Computer Program
122
Apply Program Checking Tool
406
Generate guarded verification condition and initial guard truth vector
505
Apply theorem prover
507
Update guard truth vector and combine with GVC
515
Counter Examples
509
Any counter examples associated with a candidate annotation?
510
Yes
No
Remove from program annotations whose guards are false
516
Modified Computer Program
120

FIG. 5

METHOD AND APPARATUS FOR AUTOMATICALLY INFERRING ANNOTATIONS FOR AN EXTENDED STATIC CHECKER

[0001] This application claims priority to provisional patent application entitled "Method and Apparatus for Automatically Inferring Annotations For an Extended Static Checker," Serial No. 60/251,304, filed Dec. 4, 2000, and to provisional patent application entitled "Method and Apparatus for Automatically Inferring Annotations," Serial No. 60/251,305, filed Dec. 4, 2000, both of which are incorporated herein by reference.

FIELD OF THE INVENTION

[0002] The present invention relates generally to program checking tools that automatically verify, using static checking techniques, the correctness of a computer program with respect to predefined criteria. The present invention relates particularly to an inference system that automatically annotates the computer program by iterative application of a program checking tool such as an extended static checker so as to eliminate or reduce spurious warning messages produced by the program checking tool.

BACKGROUND OF THE INVENTION

[0003] The purpose of a program checking tool is to analyze a given computer program to determine whether or not it has certain desirable properties. Program checking tools, often called program checkers, are specific examples of verification systems that can also be used to analyze hardware components, formulae, algorithms, or, more generally, behavioral designs.

[0004] A program checking tool may generate a verification condition from a given computer program. A verification condition (VC) is a logical formula that, ideally, is valid if and only if all possible behaviors of the program have the desirable properties under consideration. The program checking tool then processes the verification condition with a theorem prover.

[0005] The theorem prover should have the property that when it fails to generate a proof it generates a number of potential counter examples. The program checking tool then post-processes these counter examples into warnings that the desirable properties may not hold. A warning may be spurious; that is, it may warn about something that is not a real error, as may arise when the theorem prover does not have enough information to generate a proof.

[0006] A good program checking tool has the property that the warnings it produces are informative and easy for a designer to understand. An informative warning message should, ideally, include a characterization of each possible defect (e.g., "array index out of bounds", "timing constraint not satisfied", "race condition", "deadlock", "failure to establish invariant") and a source location in the computer program where the verification system tried, but failed, to show the absence of the defect (e.g., "line 218 of file 'ABC.source'"). If a warning message is informative and easy to understand, the designer can more easily determine whether a warning is real or spurious, and what its cause is. The designer can then act accordingly, correcting the program at the source of the problem, or ignoring the warning, possibly annotating the program so that the warning will be suppressed next time the program checking tool is run. The cost of a programming error can be greatly reduced if it is detected early in the development process.

[0007] Static checkers catch errors at compile time without executing the program and are valuable because they can be applied throughout the development cycle. A common example of a static checker is a type checker, which detects errors such as the application of a function to inappropriate argument values. Another static checker is the Compaq Extended Static Checker for Java ("ESC/Java"), which checks for additional errors that are not caught by traditional type checker systems, such as dereferencing a null pointer, indexing an array outside its bounds, or accessing a shared variable without holding its protecting lock. ESC/Java uses an underlying automatic theorem prover to precisely reason about whether or not these kinds of errors can occur.

[0008] Static checkers generally rely on the programmer to supply annotations. The computer program may be annotated by a developer to indicate aspects that may not be apparent to the checker, or to impose restraints on how the program operates, or to describe program properties such as invariants. The annotations may permit the program checking tool to find defects using a local (modular) analysis, because the annotations provide a specification of other parts of the program. In modular checking, the static program checker analyses one program module at a time, where a module may be a function, subroutine or some suitable compartment of the program. During such a modular analysis, the program checking tool verifies that the supplied annotations are consistent with the program. The presence of the annotations guides the checking process, thus making the checking problem conceptually and computationally simpler.

[0009] For example, conventional type checkers follow this modular approach and rely on type annotations to guide the type checking process. Similarly, static race detection checkers, like rccjava (Flanagan, C., and Freund, S. N., "Type-based race detection for Java," PLDI'00, *ACM SIGPLAN Notices,* 35(5):219-232, May 2000) rely on annotations describing the locking discipline. Additionally, extended static checkers like ESC/Modula-3 (Detlefs, D. L., Leino, K. R. M., Nelson, G., and Saxe, J. B., "Extended Static Checking," Research Report 159, Compaq Systems Research Center, December 1998) and ESC/Java (see www.research.compaq.com/SRC/esc/Esc.html) are modular checkers whose annotations include preconditions, postconditions, and object invariants.

[0010] The main costs in using a program checking tool, from the perspective of the programmer, comprise annotating the program, waiting for the tool to complete its analysis, and interpreting the tool's output. Often the dominant cost of using a program checking tool is annotating the program, especially for large legacy programs, because of the number of special constraints and conditions that need to be conveyed to the program checking tool via annotations.

[0011] Thus, a limitation of the modular checking approach is the burden on the programmer to supply annotations. Although programmers have grown accustomed to writing type annotations, they have been reluctant to provide additional annotations. This reluctance has been the major obstacle to the adoption of modular checkers like ESC/Java and rccjava. The burden of introducing annotations appears particularly pronounced when faced with the daunting task of applying such a checker to existing (unannotated) code bases. Preliminary experience with ESC/Java has indicated that a programmer can annotate an existing unannotated program at the rate of at most a few hundred lines per hour, though a lower rate is more usual if the programmer is unfamiliar with the code.

[0012] A new approach developed in conjunction with the present invention utilizes the warnings produced by the program checking tool itself to infer annotations and inserts those annotations directly into the program. In this way, the program checking tool functions much as a black box in the sense that its internal workings are irrelevant for the purpose of the analysis. Such an approach can be repeated iteratively in such a way as to generate a modified computer program containing many new annotations at relatively little burden to the author, but in such a way that the annotations are intelligible.

[0013] A method involving iterative modifications to a computer program obtained by using a static checker is also described in concurrently filed and commonly assigned U.S. patent application, Ser. No. ______, entitled "Method and Apparatus for Automatically Inferring Annotations," incorporated herein by reference. An algorithm for enabling annotation inference by iterative application of a static checker is described in: Flanagan, C., Joshi, R. and Leino, K. R. M., "Annotation Inference for Modular Checkers,"*Information Processing Letters,* 77:97-108 (2001), incorporated herein by reference.

[0014] Houdini is an annotation assistant that embodies this approach (see, Flanagan, C., and Leino, K. R. M., "Houdini, an Annotation Assistant for ESC/Java," SRC Technical Note 2000-003, which also appears in: Flanagan, C. and Leino, K. R. M., "Houdini, an annotation assistant for ESC/Java," in *International Symposium of Formal Methods Europe* 2001: *Formal Methods for Increasing Software Productivity,* vol. 2021 of Lecture Notes in Computer Science, 500-517. Springer, (March 2001)) to make ESC/Java more useful in catching defects in legacy code. Essentially, Houdini conjectures heuristically a large number of candidate annotations for an unannotated program, many of which will be invalid, and then repeatedly uses ESC/Java as a subroutine to verify or refute each of these annotations.

[0015] Nevertheless, a scheme involving iterative modifications to the computer program entails an overhead corresponding to the cost of making the modifications at each iteration and is therefore slow to run. It would be more convenient to modify the computer program just once, after the annotations have converged.

SUMMARY OF THE INVENTION

[0016] In summary, the present invention falls within a class of program verifiers known as static checkers and is designed to reduce the cost of annotating programs. The present invention uses the program checking tool as a black box and utilizes the warnings produced by the program checking tool itself to refute annotations. The functionality of the program checking tool is thereby leveraged, rather than being duplicated. The annotation inference module modifies the computer program, by adding candidate annotations to a computer program and then removing refuted annotations from the program. In particular the present invention uses facilities of a program checking tool such as an extended static checker to reduce the cost overhead of modifying the computer program.

[0017] Accordingly, the present invention includes a method of annotating a computer program with a least one unrefuted annotation, which begins with inserting a set of candidate annotations into the computer program to create an annotated computer program. At least one guarded verification condition is generated from the annotated computer program, wherein the guarded verification condition comprises a set of guards. Each guard in the set of guards corresponds to an annotation in the set of candidate annotations, and an initial truth value of each of the guards is set to true. A theorem prover is applied to the at least one guarded verification condition, to produce one or more counter examples. For each of the counter examples that indicates that there is an inconsistency between the computer program and at least one annotation in the set of candidate annotations, the method updates the truth value of each guard that corresponds to the at least one annotation. The applying and the updating are repeated until the theorem prover produces no counter examples that indicate that there is an inconsistency between the computer program and an annotation in the set of annotations. Finally, the computer program is modified so as to remove every annotation whose truth value has been updated, thereby creating a modified computer program that contains at least one unrefuted annotation.

[0018] The system and computer program product of the present invention implement this method of annotating a computer program.

[0019] Accordingly, the present invention further includes a computer program product for use in conjunction with a computer system. The computer program product comprising a computer readable storage medium and a computer program mechanism embedded therein. The components of the computer program mechanism include: a set of instructions for inserting a set of candidate annotations into a computer program; a verification condition generator for generating at least one guarded verification condition from the annotated computer program wherein the guarded verification condition comprises a set of guards wherein each guard in the set of guards corresponds to an annotation in the set of candidate annotations and wherein an initial truth value of each of the guards is set to true; a theorem prover for producing, from the at least one guarded verification condition, one or more counter examples; a guard truth vector updater for updating the truth value of each guard that corresponds to an annotation that corresponds to at least one of the counter examples, is inconsistent with the computer program; control instructions for iteratively applying the theorem prover and guard truth vector updater until the theorem prover produces no counter examples that indicates that there is an inconsistency between the computer program and an annotation in the set of annotations; and instructions for modifying the computer program so as to remove every annotation whose truth value has been updated thereby creating a modified computer program that contains at least one unrefuted annotation.

[0020] The present invention also includes a system for annotating a computer program with at least one unrefuted annotation. This system includes at least one memory, at least one processor and at least one user interface, all of which are connected to one another by at least one bus. The at least one processor is configured to annotate the computer program with at least one unrefuted annotation. The processor executes instructions to: insert a set of candidate annotations into the computer program; generate at least one guarded verification condition from the annotated computer program wherein the guarded verification condition comprises a set of guards wherein each guard in the set of guards corresponds to an annotation in the set of candidate annotations and wherein an initial truth value of each of the guards is set to true; apply a theorem prover to produce, from the at least one guarded verification condition, one or more counter examples; update the guard truth vector so that the truth value of each guard that corresponds to an annotation that corresponds to at least one of the counter examples, is inconsistent with the computer program; iteratively apply the theorem prover and guard truth vector updater until the theorem prover produces no counter examples that indicates that there is an inconsistency between the computer program and an annotation in the set of candidate annotations; and modify the computer program so as to remove every annotation whose truth value has been updated thereby creating a modified computer program that contains at least one unrefuted annotation.

[**0021**] In a preferred embodiment, the set of candidate annotations is derived by employing a heuristic analysis of the computer program. In an especially preferred embodiment, the program checking tool is an extended static checker.

BRIEF DESCRIPTION OF THE DRAWINGS

[**0022**] Additional objects and features of the invention will be more readily apparent from the following detailed description and appended claims when taken in conjunction with the drawings, in which:

[**0023**] **FIG. 1** is a block diagram of a programmed general purpose computer according to an embodiment of the annotation inference system of the present invention.

[**0024**] **FIG. 2** is a flow chart showing the application of the annotation inference system to a computer program in order to generate an annotated computer program.

[**0025**] **FIGS. 3A and 3B** are flow charts showing the application of a program checking tool to an annotated computer program to generate warnings.

[**0026**] **FIG. 4** is a flow chart showing an iterative method of inferring annotations in which the program is modified at each iteration.

[**0027**] **FIG. 5** is a flow chart showing a preferred embodiment of the method of inferring annotations according to the present invention.

In flow charts it is not intended that the ordering of steps as shown is necessarily the ordering that must be carried out when practicing the methods of the present invention.

DESCRIPTION OF THE PREFERRED EMBODIMENTS

[**0028**] The methods of the present invention are described with respect to a computer program and a program checking tool, but it is to be understood that the methods are equally applicable to algorithms, formulae, hardware descriptions or, more generally, behavioral designs and their respective associated verification systems.

[**0029**] Hereinafter, when using the term procedure, as used to mean a portion of a computer program, it is also assumed that the discussion can also apply to a 'class, ''module,''function,' or 'subroutine,' depending upon the computer language or system employed.

[**0030**] Referring to **FIG. 1**, the present invention may be implemented using a programmed general-purpose computer system **100**. The computer system **100** includes: (a) one or more data processing units (CPU's) **102**; (b) memory **106**, which will typically include both high speed random access memory as well as non-volatile memory (such as one or more magnetic disk drives); (c) a user interface **108** which may comprise a keyboard, mouse and/or touch-screen display; (d) a network or other communication interface **110** for communicating with other computers as well as other devices; and (e) one or more communication busses **104** for interconnecting the CPU(s) **102**, memory **106**, user interface **108**, and network interface **110**.

[**0031**] The computer system's memory **106** stores procedures and data, typically including:

[**0032**] an operating system **112** for providing basic system services;

[**0033**] application programs **114**, such as user level programs for viewing and manipulating images;

[**0034**] authoring tools **116**, for assisting with the writing of computer programs;

[**0035**] a computer program **118** possibly containing some annotations, to be analyzed by an annotation inference system **130**;

[**0036**] a modified computer program **120** that is the product of applying annotation inference system **130** to computer program **118** and which contains at least one unrefuted annotation;

[**0037**] an annotated computer program **122** that results from inserting a candidate set of annotations into computer program **118**;

[**0038**] optionally, an intermediate modified computer program **124** that results each time one or more annotations from the candidate set is refuted by the annotation inference system; and

[**0039**] an annotation inference system **130** for automatically inserting annotations into computer program **118**.

[**0040**] The annotation inference system **130** preferably includes:

[**0041**] an annotation inference module **132**, also called an annotation assistant, which is the main procedure of the annotation inference system and controls its overall operation; and

[**0042**] a program checking tool **150** that, when applied to a computer program **118**, produces zero or more warnings.

[**0043**] Together, the elements of the annotation inference module **132**, or annotation assistant, along with those of program checking tool **150**, may be referred to as an annotation inference system **130**. In particular, the annotation inference module is able to control the running of the program checking tool.

[**0044**] The annotation inference module **132** preferably includes:

> [**0045**] a heuristic annotation generator **134** that parses computer program **118** and suggests one or more annotations;
>
> [**0046**] a counter example mapper **136** that maps counter examples **158** into annotation modifications **138**;
>
> [**0047**] a guarded verification condition **142** that comprises a verification condition with additional guards corresponding to annotations;
>
> [**0048**] a guard truth vector **144** that contains the set of truth values for all the guards in guarded verification condition **142**;
>
> [**0049**] a guard truth vector updater **146** that updates the guard truth vector **144** by setting to false the truth value of the guards corresponding to refuted annotations; and
>
> [**0050**] optionally, one or more annotation modifications **138** corresponding to annotations from the candidate set that are refuted by the annotation inference system; and
>
> [**0051**] a program updater **140** that inserts or removes annotations from computer program **118** according to suggestions associated with the annotation modifications in order to produce a modified program **120** that contains at least one unrefuted annotation.

[**0052**] The program checking tool **150** is preferably an extended static checker and preferably includes:

> [**0053**] a verification condition generator **152** for converting a program into a logical equation called a verification condition **162**;
>
> [**0054**] a theorem prover **154** that attempts to prove or refute the verification condition **162**;
>
> [**0055**] an analysis module **156** that converts counter examples into warning messages;
>
> [**0056**] zero or more counter examples **158**;
>
> [**0057**] zero or more warning messages **160**; and
>
> [**0058**] at least one verification condition **162** corresponding to a procedure in computer program **118**.

[**0059**] Other configurations of the various items in memory **106**, as described hereinabove, are consistent with the operation of the present invention.

Overview of Operation of an Annotation Inference System and Program Checking Tool

[**0060**] The general scheme in which the present invention operates is presented in **FIG. 2**. A computer program **118** is provided to an annotation inference system **130** which produces a modified computer program **120** as output. The modified computer program **120** preferably contains one or more unrefuted annotations that were not present in the original computer program **118** and which have been supplied by the annotation inference system. The original computer program **118** may itself already contain some annotations prior to application of the annotation inference system.

[**0061**] The annotation inference system of the present invention utilizes a program checking tool **150** that checks computer programs for defects, as shown in **FIG. 3A**. The tool itself takes as input an annotated computer program **122** containing one or more annotations. These annotations are initially the original set of annotations, Ann. The annotations may indicate properties that are expected to hold or are supposed to hold at various program points and therefore help the program checking tool **150** check the program. The program checking tool is applied to the computer program **122** and zero or more warnings **160** are produced, possibly as a result of analyzing one or more counter examples **158**.

[**0062**] In a preferred embodiment, the program checking tool **150** is an extended static checker (ESC) that operates under the control of the annotation inference module **132**. Examples of extended static checkers are ESC/Java and ESC/Modula-3. For the purposes of the present invention it suffices that the program checking tool generates a verification condition and comprises a theorem prover, though any program or programs that provide access to a verification condition generator and a theorem prover would suffice.

[**0063**] **FIG. 3B** shows the procedure of **FIG. 3**A, augmented to illustrate the internal workings of the program checking tool. The program checking tool uses a two-step approach to verifying the computer program. In a first step it converts each procedure in annotated computer program **122** into a corresponding verification condition **162**. For the purposes of the present invention, when discussing the conversion of a program into a verification condition, it is assumed that the verification condition can itself comprise more than one separate verification conditions corresponding to one or more procedures in the computer program. A verification condition is a predicate whose universal truth is tested by a theorem prover. Each annotation may appear in the verification condition zero or more times, possibly in some modified form. In a second step, one or more of the verification conditions are passed to an automatic theorem prover **154**. The theorem prover refutes a verification condition if it contains an incorrect annotation, i.e., if there is a possible error in the corresponding procedure. One or more counter examples **158** are output from the theorem prover, and can be transformed into one or more warning messages **160** suitable for interpretation by a user.

[**0064**] Counter examples are generally mathematical equations or contexts that are consistent with one another but indicate conditions that are contrary to one or more verification conditions. A counter example may be a simple mathematical expression of the form "$x<0$" (which would be a counter example to a proposition such as "$x>10$") or may be more complicated. Counter examples, as produced by a theorem prover, may be intelligible to a user but are preferably transformed into warning messages that indicate specific points in the computer program at which a specific condition is found not to hold, and are more readily understood by a user. Counter examples may, however, be readily parsed, analyzed or otherwise processed by software such as analysis module **156**, or other software modules of the present invention.

Generation of Verification Conditions

[**0065**] In a preferred embodiment, the transformation of the annotated computer program **122** into the verification condition itself occurs via a two-stage process (as described in: K. R. M. Leino, J. B. Saxe and R. Stata, "Checking Java programs via guarded commands,"*SRC Technical Note* 1999-002, Compaq Computer Corporation, (May 21, 1999), also available in *Formal Techniques for Java Programs,* Workshop proceedings, Ed. B. Jacobs, et al., Technical Report 251, Femuniversität Hagen, (1999), incorporated herein by reference). The computer program source statements are first converted into an intermediate language, and then weakest precondition operators are used to process the intermediate-language statements into verification conditions (as described in U.S. Pat. No. 5,987,252 which is hereby incorporated herein by reference).

[**0066**] In a preferred embodiment, the intermediate form of the computer program is expressed in a particularly simple programming language that has no procedure call statements. Instead, the conversion to intermediate language replaces each call by its meaning according to the called procedure's pre- and postcondition annotations. In a preferred embodiment, the intermediate programming language utilizes guarded commands. For a description of guarded commands, see E. W. Dijkstra, *A Discipline of Programming,* Prentice-Hall, (1976). Other examples of guarded commands derived from Dijkstra are described elsewhere (see, e.g., G. Nelson, "A Generalization of Dijkstra's Calculus",*ACM Transactions on Programming Languages and Systems,* 11(4): 517-561, (1989), incorporated herein by reference). Accordingly, it will be understood by one of skill in the art that the methods of the present invention are not limited to any particular set of guarded commands but are applicable to Dijkstra's original commands and many other variations thereof. The conversion of Java programs to a set of guarded commands is described in: K. R. M. Leino, J. B. Saxe and R. Stata, "Checking Java programs via guarded commands,"*SRC Technical Note* 1999-002, Compaq Computer Corporation, May 21, 1999.

[**0067**] The intermediate language contains assert and assume statements that bear labels, so as to keep track of whether the statement originated in the source or was generated on behalf of some annotation, and if so, which one. The labels of assert statements are used by the program checking tool to keep track of which annotations are to be refuted.

[**0068**] The intermediate form of the program is processed by the verification condition generator **152** to produce a verification condition **162** for the program. The verification condition (VC) is a first order logical formula built up from the constants "false" and "true," atomic boolean program expressions such as equality and inequality relations between program variables, the usual boolean connectives, and universal quantification. Additionally, the formula can be labeled by an annotation or program location, yielding a labeled formula. While the labels do not change the meaning of the underlying formula, they provide information to the subsequent operation of the program checking tool.

[**0069**] In a preferred embodiment, the verification condition generator **152** can produce a guarded verification condition **142** for the program, as described hereinbelow, without first producing a verification condition.

[**0070**] In a preferred embodiment, the logical formula is expressed as a "weakest precondition." The weakest precondition of a statement, S, with respect to a postcondition R is the formula that characterizes those initial states from which the execution of S does not go wrong and terminates only in states satisfying R. Methods of expressing weakest preconditions for statements expressed in guarded commands are given by Dijkstra (see, E. W. Dijkstra, *A Discipline of Programming,* Prentice-Hall, (1976)). The logical formula is typically represented as a tree of sub-expressions. Various subsets and combinations of the sub-expressions must be conclusively proved to be true for all possible program conditions.

Application of the Theorem Prover

[**0071**] The verification condition, VC, is passed to the theorem prover **154** whose job is to evaluate the sub-expressions of the VC, for all possible program conditions, to determine which ones (if any) it cannot conclusively prove to be true. Failure to prove sufficient combinations of sub-expressions to always be true means that one or more of the pre-conditions or postconditions required for proper operation of the program is not satisfied, or may potentially not be satisfied.

[**0072**] Even if a verification condition, VC_f, for a procedures f, is found to be not valid, thus indicating that an invocation of f may violate some annotation, in order to indicate which annotation is violated it is preferable to introduce some extra machinery. Identifying invalid annotations can utilize a mechanism of exposing a labeled subformula in a VC. This is accomplished by defining a suitable function, expose, such that a formula, R, refutes an annotation α, if expose(α,VC) is not valid. A definition of expose is given with a discussion of mathematical formalisms, hereinbelow.

[**0073**] When it is unable to prove the truth of the VC, the theorem prover ideally produces one or more counter examples **158**. These counter examples can be processed by an analysis module **156** and output as warning messages **160**. A discussion of counter examples and their mapping into warnings is outside the scope of this document (but a discussion may be found in commonly assigned pending U.S. patent application Ser. No. 09/754,890, entitled, "System and Method for Verifying Computer Program Correctness and Providing Recoverable Execution Trace Information," filed Jan. 5, 2001, incorporated herein by reference). The program checking tool may produce at least two kinds of warnings. Each counter example contains sufficient information for the program checking tool to figure out whether it constitutes a warning of the first kind or a warning of the second kind. A counter-example can also contain labels corresponding to annotations.

[**0074**] Warnings of a first kind are warnings about possible misapplications of primitive operations of the programming language. For example, these warnings concern potential run-time errors, such as dereferencing a null pointer, and indicate that the computer program may not work. Such warnings are denoted 'W0' in commonly assigned, concurrently filed U.S. patent application Ser. No. ______, entitled "Method and Apparatus for Automatically Inferring Annotations."

[0075] Warnings of a second kind alert a user about inconsistencies between the program and particular annotations. Such warnings are denoted 'W1' in commonly assigned, concurrently filed U.S. patent application Ser. No. ______, entitled "Method and Apparatus for Automatically Inferring Annotations." Warnings of the second kind occur if the program checking tool is not able to verify the program property claimed by an annotation, i.e., the annotation is inconsistent with the program. For example, such a warning is generated if preconditions of a procedure are not satisfied at the call site of the procedure. The annotation assistant interprets such warnings as refuting incorrect guesses in the candidate annotation set. The annotations that give rise to such warnings are preferably removed from the computer program text by the annotation inference system. Such annotations are sometimes called "refuted annotations." The approach to refuting annotations employed by the annotation inference module of the present invention is described hereinbelow.

[0076] The method of the present invention finds the largest subset of the original annotations that is valid for the program. The algorithm employed starts with the original set of annotations, Ann, and removes annotations from it until a valid subset is reached.

[0077] The original set of annotations may comprise annotations originally present in the program as well as a candidate set of annotations heuristically guessed and inserted into the program by the annotation inference system **130**. For the purposes of the methods of the present invention, a subset of the annotations corresponding to the candidate set of annotations are refuted and subsequently removed.

[0078] In a preferred embodiment, the algorithm maintains a work list, W, that contains the procedures that are still unchecked with respect to the current set of annotations. When checking a procedure, f, from W, any candidate annotations in the current set that are not valid for f, are removed from the current set and the work list is extended with the procedures that assume any of the refuted annotations. The algorithm terminates when the work list becomes empty and at this point the current set of annotations becomes the largest valid subset of Ann.

[0079] Since removing one annotation may cause subsequent annotations to become invalid, this check-and-refute cycle iterates until a fixed point is reached. The process terminates because, until a fixed point is reached, the number of remaining candidate annotations is strictly decreased with each iteration. The resulting annotation set is clearly a subset of the original set, and is valid with respect to the static checker, that is, the static checker does not refute any of its annotations. The inferred annotation set is in fact a maximal valid subset of the candidate set. Furthermore, this maximal subset is unique. For a proof of these properties, and also a more efficient version of the basic algorithm presented here, see Flanagan, C., et al., "Annotation Inference for Modular Checkers,"*Information Processing Letters,* 77:97-108 (February, 2001).

[0080] As an example of why the refutation of one annotation may cause subsequent annotations to become invalid, consider a candidate annotation, x>0, as a precondition for procedures p and q wherein procedure p calls procedure q. If the program checking tool finds that, elsewhere in the program, procedure r sets x=−5 prior to calling p, then the precondition on p is removed. On a subsequent application of the program checking tool, a warning will be generated for the precondition on x as applied to q.

Iterative Application of a Theorem Prover in Which a Computer Program is Successively Updated

[0081] A method of using the annotation inference module to modify annotations in a computer program, without exploiting the advantages of the present invention, is described for comparison purposes with respect to **FIG. 4**. The input is the computer program **118**. According to this method, the annotation inference module starts by heuristically constructing from the program a finite candidate set of annotations, step **402**, though the program may also contain manually inserted annotations. Methods of heuristically generating annotations are described hereinbelow. Ideally, the candidate set is sufficiently large to include all annotations that may be useful when applying the program checking tool to the program at step **406**. The annotation inference module inserts the candidate annotations into the program, step **404**, to produce an annotated computer program, **122**. Then, the annotation inference module applies the program checking tool to the annotated program, step **406**, thereby producing warnings **160**. In practice, at step **406**, the program checking tool translates the annotated computer program into a verification condition, to which the theorem prover is applied, to generate counter examples. Warnings are generated from the counter examples using an analysis module **156**, shown in **FIG. 1**. Invocation of the program checking tool thus produces warnings about portions of the program that violate some of the given annotations. The annotation inference module inspects all warnings of the second kind produced by the tool, step **410**, and, if there are no such warnings, provides the user with a modified computer program **120**. If there are warnings of the second kind that are suitable for mapping into annotation modifications the counter example corresponding to each such warning is mapped into an annotation modification, step **412**. The annotation inference module interprets such warnings as identifying incorrect annotation guesses in the candidate set. The annotation inference module acts on these annotation modifications so that any candidate annotation mentioned in these warnings is removed from the modified computer program at step **414** by the program updater, thereby producing an intermediate modified computer program **416**. In this sense, an invocation of the program checking tool has the effect of refuting some number of candidate annotations. The program checking tool is then applied again to the intermediate modified computer program **416** at step **406**. Steps **406**, **410**, **412**, and **414** represent a loop that is repeated until the program checking tool no longer produces any warnings of the second kind that involve a candidate annotation.

[0082] The net effect of the loop is to remove as many incorrect candidate annotations as are possible while retaining those that are not inconsistent with one another. Thus, the annotations remaining upon termination comprise a correct subset of the candidate set. This subset will be the greatest subset whose validity can be established consistent with other members of the candidate set.

[0083] In pseudo-code, this method of applying the annotation inference module can be expressed as follows:

```
use heuristics to generate candidate annotation set;
repeat
    invoke program checking tool to refute annotations;
    remove the refuted annotations;
until quiescence
```

[0084] In practice, the program checking tool can be an extended static checker which is applied to the program, and any annotation deemed incorrect by the extended static checker is removed at each iteration.

[0085] After modified computer program **120** is produced, the program checking tool is applied to it to produce a final set of counter examples that are mapped to warning messages. The warning messages are presented to the user.

Overview of the Method of the Present Invention

[0086] When refuting annotations, an annotation inference module using the algorithm described hereinabove may need to invoke the program checking tool to check each procedure a large number of times. Each invocation of the checker would involve both generating a procedure's verification condition and checking its validity. However, the former operation need not be carried out every time since the structure of a verification condition is derived from the code of a procedure, which remains constant. Accordingly, the present invention seeks a way of updating the current set of annotations without recreating the VC at every iteration.

[0087] The present invention solves the problem of computational overhead during the iterations by calling the theorem prover directly and by introducing annotation guards into the VC. By enabling annotations to be removed from the verification condition without having to modify the text of the original program, and without having to recreate the verification condition at each iteration, computationally expensive processes are avoided.

[0088] According to the method of the present invention, an improvement to an annotation assistant that employs a program checking tool such as an extended static checker is realized by creating a modified form of the verification condition called a guarded verification condition (GVC). A GVC includes special boolean variables called annotation guards, one for each annotation. The collection of annotation guards is called the guard truth vector. By using guarded verification conditions, it is possible to create the verification condition for each procedure just once, and then to re-run the theorem prover on the verification condition multiple times, using the annotation guards to record the current state, i.e., which annotations have been refuted. Counter examples produced by the theorem prover are used to update entries in the guard truth vector that correspond to annotations in the candidate set. This approach avoids having to recreate the verification condition at each iteration, as done in earlier algorithms such as employed in the earlier versions of Houdini (see *SRC Technical Note* 2000-003), and thus the new approach is significantly faster than previous approaches.

Using Guarded Verification Condition Expressions

[0089] Referring to **FIG. 1**, according to the method of the present invention, the program checking tool **150** includes a verification condition generator **152** for converting a program into a logical equation called a verification condition **162**. The verification condition is converted by the annotation inference module into a guarded verification condition **142**. The guarded verification condition (GVC) includes a number of "guards," or "guard variables." The guards of the guarded verification condition correspond to the program annotations. The notation g_α is used to denote the annotation guard, or guard variable, for some annotation a. The guarded verification condition is created in the same manner as a conventional verification condition, except that where a predicate P resulting from an annotation α would normally be inserted into a conventional verification condition, the implication "$(g_\alpha \rightarrow P)$" is instead inserted into the GVC. Each of the guards is assigned a truth value, and the set of truth values for all the guards in the verification condition is called the guard truth vector **144**. Thus, by fixing the annotation guard g_α to be true, the predicate resulting from the annotation is present in the GVC. Alternatively, if g_α is false then the implication simplifies to true, (because "false implies something" can be anything at all and is thereby 'true') and the predicate can be considered to be absent from the GVC. Initially all of the annotation guards are set to true.

[0090] The program checking tool **150** includes a theorem prover **154** that attempts to prove or refute the guarded verification condition **142**, where the truth value of each of the guards in the verification condition is set in accordance with the current state of the guard truth vector **144**. A guard truth vector updater **148** updates the guard truth vector **144** by setting to "false" the truth value of the guards corresponding to refuted annotations, if any.

[0091] The following translation is defined for generating guarded verification conditions. The guarded weakest precondition translation $gwp \in Stmt \times Formula \rightarrow Formula$ is shown in Table 1, hereinbelow.

TABLE 1

Guarded Weakest Preconditions of Exemplary Guarded Commands

S	gwp(S,R)	
$v: = e$	$R(v: = e)$	
assert α: e	$(g_\alpha \Rightarrow (\text{label } \alpha{:}e)) \wedge R$	if α corresponds to a check that would give rise to a warning of the second kind.
	$e \Rightarrow R$	if α corresponds to a check that would give rise to a warning of the first kind.
assume α: e	$(g_\alpha \Rightarrow e) \Rightarrow R$	
$S_1; S_2$	$gwp(S_1, gwp(S_2,R))$	
$S_1 \Box S_2$	$gwp(S_1, R) \wedge gwp(S_2, R)$	
var w in S end	$< w :: gwp(S, R)>$	provided no variable w occurs free in R

[0092] In Table 1, S is an intermediate language statement, such as a guarded command. The guarded weakest precondition of S is given with respect to an annotation, $\alpha \in Ann$, and a post-state predicate, R. The variable g_α is the guard variable associated with annotation, α. In an expression such as $g_\alpha \rightarrow e$, substituting "true" for g_α makes it equivalent to just e, while substituting "false" makes the expression equivalent to "true." An expression gwp(S, R) characterizes the pre-states from which executions of S do not go wrong and terminate only in states satisfying R.

[0093] The analogous expressions to those in Table 1 for the weakest precondition, wp, can be found in: Flanagan, C., Joshi, R., and Leino, K. R. M., "Annotation Inference for Modular Checkers,"*Information Processing Letters,* 77:97-108 (February, 2001). Note that the definition of gwp differs from that of the weakest precondition only in "assert" and "assume."

[0094] In a preferred embodiment, a single "template condition" is generated for every procedure at the beginning of the annotation inference process. When a procedure needs to be checked, its template condition is converted to an appropriate verification condition by replacing the parts related to the refuted annotations with "true" and leaving the parts related to the remaining annotations unchanged.

[0095] To convert a guarded verification condition into an ordinary verification condition, "false" is substituted for the guard variables associated with refuted annotations and "true" is substituted for the remaining guard variables. This is formalized by the function dropGuards, which maps a formula and a set of annotations to a formula. The function dropGuards is preferably defined as follows, wherein A is a set of annotations:

dropGuards(g_α, A) = true	if $\alpha \in A$
dropGuards(g_α, A) = false	if α A
dropGuards(e, A) = map (<λF:: dropGuards(F,A)>, e)	if e is any other expression than a guard variable

[0096] In this definition of dropGuards, as would be understood by one of ordinary skill in the art, the map expression maps the function dropGuards over the operators in expression, e, and F is a dummy variable. It is understood that other definitions and implementations of dropGuards that perform substantially the same function are compatible with the methods of the present invention.

[0097] A preferred embodiment of the method of the present invention is described with respect to **FIG. 5**. The input is annotated computer program **122** preferably created according to steps **402** and **404** of **FIG. 4**. In step **402** the annotation inference module heuristically constructs from a computer program **118** a finite candidate set of annotations. Ideally, the candidate set is sufficiently large to include all annotations that may be useful when applying the program checking tool to the program. The annotation inference module inserts the candidate annotations into the program, step **404**, to produce annotated computer program **122**. Alternatively, annotated computer program **122** may contain annotations inserted manually by a user. Such annotations may themselves be inconsistent with the computer program and may therefore be suitable for refutation, if the user so requires. Thus it is compatible with the methods of the present invention that a user could stipulate that certain annotations are to be refuted and that others are to be preserved.

[0098] The annotation inference system then applies the program checking tool **122** to the annotated computer program to produce a verification condition, which is converted into a guarded verification condition, step **505**. In a preferred embodiment, program checking tool **122** produces a guarded verification condition directly from the annotated computer program. Associated with the guarded verification condition is a guard truth vector whose truth values are initially set to true. The annotation inference module iteratively applies the theorem prover **146** to the guarded verification condition. At each subsequent iteration of the main loop of the procedure, the theorem prover is re-executed (step **507**) without regenerating the verification condition. Instead, the guarded verification condition is evaluated in accordance with the current state of the guard truth vector.

[0099] Each application of the theorem prover, step **507**, produces zero or more counter examples **509**. In a preferred embodiment, in order to associate a counter example with a heuristic annotation, a counter example contains a label. The annotation inference module is able to parse the counter example in such a way that it can check whether the label is associated with an annotation from the candidate set. Ways to insert labels into counter examples generated by a theorem prover are described in commonly assigned pending U.S. patent application Ser. No. 09/754,890, entitled, "System and Method for Verifying Computer Program Correctness and Providing Recoverable Execution Trace Information," filed Jan. 5, 2001, incorporated herein by reference. The annotation inference module inspects counter examples that correspond to warnings of the second kind produced by the program checking tool, step **510**, and, if there are no such counter examples removes from the computer program annotations whose annotation guards are false, step **516**, and provides the user with a modified computer program **120**.

[0100] If there are counter examples corresponding to warnings of the second kind, the guard truth vector updater updates the guard truth vector at step **515** so as to mask any candidate annotation mentioned in these counter examples. That is, at step **515** the guard truth vector is updated so as to set to "false" the truth value of each guard that corresponds to a refuted candidate annotation. The updated guard truth vector is combined with the guarded verification condition without regenerating the verification condition. The theorem prover is then applied again to the GVC at step **507**. Steps **507**, **510** and **515** are repeated until the theorem prover no longer produces any counter examples corresponding to warnings of the second kind and which involve a candidate annotation. At such time, the program updater removes from the computer program annotations whose annotation guards are false, step **516**, and provides the user with a modified computer program **120**.

[0101] In pseudo-code, the preferred embodiment of the annotation inference system can be expressed as follows:

```
Use heuristics to generate a candidate annotation set;
Create a GVC for each procedure in the program;
Set the initial value of each annotation guard to true;
do
    Run the theorem prover on each GVC, using
        the current values of the annotation guards;
    for each annotation refuted by the theorem prover
        Set the annotation guard to false;
    end
until quiescence;
```

[0102] The algorithm pre-computes a guarded verification condition of every procedure in advance and, in a preferred embodiment, applies dropGuards to convert a guarded verification condition into a verification condition whenever a procedure needs to be checked. This algorithm is efficient because in practice the application of dropGuards is much faster than the re-generation of a verification condition from the program source.

[**0103**] After iterations have converged, the annotation inference module calls the program updater in order to delete from the program those annotations that have been refuted thereby producing a modified computer program.

[**0104**] Then, the final step in the algorithm is to run the program checking tool one more time to identify potential run-time errors in the modified program. The counter examples are mapped to warning messages. These warnings are then presented to the user, and are used to identify defects in the program.

[**0105**] This algorithm works also for recursive methods. The candidate preconditions of a recursive method will be refined (by removing refuted preconditions) until the resulting set of preconditions holds at all call sites of the method, both recursive and non-recursive call sites.

[**0106**] By analyzing the dependencies between annotations and GVC's, it is also possible to modify the algorithm described hereinabove to avoid applying the theorem prover to every GVC at each iteration. The application of a similar modification to the Houdini algorithm is described in Flanagan, C., Joshi, R., and Leino, K. R. M., "Annotation Inference for Modular Checkers,"*Information Processing Letters,* 77:97-108 (February, 2001), incorporated herein by reference; one of ordinary skill in the art would be able to apply such a modification to the method of the present invention.

[**0107**] The most computationally intensive parts of the method of the present invention are its validity checks i.e., in refuting annotations. However, if multiple processors are available, it is possible to distribute these checking tasks across the available processors, so that many procedures can be checked simultaneously.

Work List Ordering Heuristics

[**0108**] The method of the present invention, as described hereinabove, is independent of how the procedures in the work list are chosen. Nevertheless, how this choice is made can have a significant impact on performance of the method. Some heuristics for ordering the procedures in the work list can usefully be employed. The methods of the present invention, wherein the verification condition is not repeatedly regenerated facilitates the use of such heuristics.

[**0109**] One category of heuristics comprises the "fastest-first" and "slowest-first" heuristics. Consider a program containing a procedure f_0 with a precondition α and containing two procedures f_1 and f_2 that each calls f_0 without establishing α. In such a scenario, analyzing either f_1 or f_2 will uncover the invalid precondition α, and the overall performance may be improved by preferring the procedure with the "faster" verification condition. Clearly, it is not possible to avoid analyzing procedures with "slow" verification conditions completely: sooner or later every procedure in the work list must be checked. Nevertheless, using this heuristic, it is hoped to reduce the number of slow checks.

[**0110**] A "fastest first" ordering heuristic is implemented by timing each verification task and associating with each procedure the amount of time it takes to check its verification condition. When the algorithm selects the next procedure for verification, it chooses the one that took the least time the last time it was analyzed.

[**0111**] A different strategy is to order the jobs by "slowest first." This heuristic may be useful in a multi-processor setting, since it may allow slow jobs to get a "head start."

[**0112**] The "no overlap" heuristic tries to avoid superfluous analysis, as can occur in a multiple-processor environment. For example, when running the distributed algorithm on a large test case, most of the time at least two processors were assigned the same procedure. While seemingly contradictory, this situation is actually possible and likely to occur. It arises when a processor i is analyzing a procedure f while another processor refutes some annotation that is assumed by f The algorithm then reinserts f into the work list and can assign it to an idle processor j before processor i finishes its verification of f.

[**0113**] One preemptive approach to implement a "no overlap" heuristic is to abort the older verification task since it is subsumed by the new one. (By monotonicity of an extended static checker, as described hereinbelow, the annotations refuted by the older task would also be refuted by the newer task.) This strategy may be profitable if many of the annotations that the older task will refute have already been refuted by other jobs.

[**0114**] Another, non-preemptive, approach is to not pick procedures that are currently being checked. This strategy may be profitable, for example, if the verification of f spends a lot of time before it starts analyzing those annotations that are not in the eventual fixpoint.

Candidate Annotations

[**0115**] The candidate annotation set is a finite set generated from the program text using heuristics, specific to the program checking tool, about what annotations are possible and/or are likely to be applicable to the program.

[**0116**] Ideally, the candidate set of annotations includes all annotations that may be useful in determining the program's correctness. However, it is also desirable to keep the candidate set reasonably small because the running time of the tool is closely related to the number of candidate annotations. Furthermore, for correctness reasons, all candidate annotations that apply to the program's entry point are required to hold at the program's initial state.

[**0117**] Methods of devising a set of candidate annotations are described in concurrently filed and commonly assigned U.S. patent application, Ser. No. ______, entitled "Method and Apparatus for Automatically Inferring Annotations," and also Flanagan, C., and Leino, K. R. M., "Houdini, an Annotation Assistant for ESC/Java," SRC Technical Note 2000-003, both of which are incorporated herein by reference. In general, examples of candidate annotations include preconditions or postconditions and relate values of program variables to certain interesting constants, such as −1, 0, 1 and constant dimensions in array allocation expressions in the same procedure.

Mathematical Formalisms

[**0118**] Some questions arise about the correctness of the annotation assistant. For example, whether or not the annotation assistant terminates with a unique answer; whether or not the order in which the checker is invoked on the various parts of the program matters; whether the checker needs to be applied to all parts of the program or the verification condition on every iteration; and upon which properties of the checker the annotation assistant relies. Such details are also to be found in: Flanagan, C., Joshi, R., and Leino, K. R. M., "Annotation Inference for Modular Checkers,"*Information Processing Letters,* 77:97-108 (February, 2001), incorporated herein by reference.

[0119] These issues can be addressed formally, adopting the following notational conventions. The power set of X is written PX. Following Dijkstra (Dijkstra, E. W., and Scholten, C. S., *Predicate Calculus and Program Semantics,* Texts and Monographs in Computer Science, Springer-Verlag, 1990), a left-associative infix "." (binding stronger than any other operator) is used to denote function application. The expression {x|r.x::t.x} denotes the set of terms of the form t.x for all x satisfying the range expression r.x. For Q denoting ∀, ∃, or any associative operator that is symmetric on the elements of {x|r.x::t.x} (for example, the union operator, ∪), the expression (Q x|r.x::t.x) denotes the application of Q to the elements of {x|r.x::t.x}. If the range expression is true, the "|true" may be omitted.

[0120] The Extended Static Checker, ESCϵProc×P Ann→P Ann, is defined by the equation:

$$ESC(p,A)=\{a|a\epsilon A\wedge[\text{expose}(a,\ VC(p,\ A)\ )]::\ a]$$

[0121] where VC is a verification condition and the item in square brackets is the validity testing operator. The invocation ESC(pA) returns the set of annotations in A not refuted by p.

[0122] The function exposeϵAnn×Formula→Formula, wherein Formula represents the syntactic class of formulae from which a VC can be composed, is defined by:

$$expose(a,\ (\text{label } b{:}e))=\begin{cases} e, & \text{if } b=a \\ \text{True}, & \text{otherwise} \end{cases}$$

[0123] expose(α, R)=map((λQ:: expose(a,Q)), R) if R is not a labeled formula.

[0124] In the definition of expose, the argument (label b:e) is a labeled formula such that formula e is labeled by an annotation or program location, b. Thus, a formula R refutes an annotation, α, if expose(α,VC) is not valid.

[0125] A modular checker checks a program one part at a time. The parts of the program on which the checker operates are referred to as "units of checking," or simply as "units." For some checkers, a unit of checking may be a routine such as a procedure, method, or constructor. For other checkers, a unit may be a larger construct such as a module, package, or class. Let Unit denote the set of possible units of checking. The internal structure of these units is of no concern: it is simply assumed that a program P⊂ Unit is a finite set of units and that a program checking tool, or checker, C, can check these units.

[0126] While checking each unit, the checking tool relies on annotations specifying properties of the other units in the program. Ann is used to denote the set of possible annotations, and whenever the program checking tool C is invoked on a unit *f* in Unit, a set of annotations A⊂ Ann is preferably also provided.

[0127] Warnings of the second kind, as described hereinabove, indicate annotations that should be refuted. During the checking process, the program checking tool may discover that the unit *f* is not consistent with some annotation in A (for example, *f* may be a procedure that fails to ensure one of its postconditions). In this case, the checker refutes the annotation. To simplify the analysis, the checker is formalized to be a function that returns the set of annotations in A that the checker fails to refute:

$$C\text{: Unit}\times P\text{Ann}\rightarrow P\text{Ann}. \tag{1}$$

[0128] The annotation inference module assumes two underlying properties of the program checking tool. The first property is that the set of annotations returned by the tool is a subset of those to which the tool is applied:

$$(\forall f,\ A|f\epsilon\ \text{Unit}\ \wedge A\subset\ \text{Ann}::\ C.fA\subset A). \tag{2}$$

[0129] The second property is that the program checking tool satisfies the following monotonicity property:

$$(\forall f|f\epsilon\ \text{Unit}::\ C.f\ \text{is monotonic}). \tag{3}$$

[0130] Intuitively, if an invocation of the program checking tool does not refute a particular annotation, then passing additional annotations to the tool does not cause that same annotation to be refuted either.

[0131] For convenience, C can also be overloaded ("lifted") to apply to sets of units: for any set F⊂ Unit,

$$C.F.A=(\cap f|f\epsilon F::\ C.f.A)\cap A \tag{4}$$

[0132] Properties (2) and (3) for a program checking tool imply analogous properties for the lifted checking tool. Furthermore, for any unit *f* that is an element of a set of units F, and set of annotations A, the following hold:

$$f\epsilon F\rightarrow C.F.A\subset C.f.A \tag{5}$$

$$f\epsilon F\wedge C.F.A=A\rightarrow C.f.A=A \tag{6}$$

[0133] Property (5) means that applying the checker to a larger set of code F increases the opportunity for refuting annotations in A. Thus, the set of unrefuted annotations C.F.A is necessarily a subset of the set of unrefuted annotations C.*f*.A.

[0134] Accordingly, it is said that an annotation set A is valid for a program P if C.P.A=A, that is, if program checking tool C does not refute any of the annotations in A. It follows from properties (2) and (3) that validity is closed under union. Hence, for any program P and annotation set A, there is a unique greatest subset of A that is valid for P.

[0135] An annotation assistant is a program that, for a given (finite) candidate annotation set G and a program P, computes the greatest subset of G that is valid for P. Formally, an annotation assistant computes a set B such that:

$$\text{B}\subset\text{G} \tag{7}$$

$$\text{C.P.B=B} \tag{8}$$

$$(\forall X|X\subset G\wedge C.P.X=X::\ X\subset B) \tag{9}$$

[0136] The following program implements an annotation assistant.

```
B:=G;
```

[0137] while C.P.B!=B do

[0138] choose X such that C.P.B⊂X⊂B;

B:=X;

[0139] end

[0140] The body of this loop picks a set X that satisfies the given range expression and then sets B to X. The loop terminates when no such X exists.

[0141] The program satisfies the specification of an annotation inference module. It is not hard to prove, using property (2), that properties (7) and (9) together are a loop invariant. By property (2), the negation of the loop guard is property (8). Termination follows from variant function |B|, which is strictly decreased by the loop body.

[0142] Note that this program can remove from B any annotation that C.P.B refutes; it need not contract B to C.P.B itself. Thus refuted annotations can be removed from B in any order.

[0143] Accordingly, the two properties (2) and (3) of the program checking tool imply that the basic annotation inference algorithm converges on a unique fixed-point, regardless of the order in which annotations are refuted and removed.

EXAMPLES

Example 1

Use of Verification Conditions

[0144] The program checking tool used in examples 1 and 2 is an extended static checker, ESC/Java, a tool for finding common programming errors in Java programs. ESC/Java takes as input a Java program, possibly annotated with ESC/Java light-weight specifications, and produces as output a list of warnings of possible errors in the program. Because of its static and automatic nature, its use is reminiscent of that of a type checker. However, ESC/Java is powered by a more precise semantics engine than most type checkers and uses an automatic theorem prover.

[0145] ESC/Java performs modular checking: every routine (method or constructor) is given a specification. ESC/Java checks that the implementation of each routine meets its specification, assuming that all routines called meet their specifications. The specification comes from user-supplied annotations. ESC/Java does not trace into the code of a callee, even if the callee code is also given to the tool to be checked. By performing modular checking, ESC/Java can be applied to a single class, or even a routine, at a time, without needing the entire program.

[0146] To demonstrate the operation of the embodiment that utilizes verification conditions and, in Example 2, guarded verification conditions, consider an example of a computer program 120 that comprises two modules, "main" and "timestwo," shown in Table 2:

TABLE 2

Example Program for Demonstrating
Verification Conditions with and without Guards

```
void main() {
    int x = 5;
    int y = timestwo(x);
    //@ assert y >= 0;
}
int timestwo(int n) {
    return 2*n;
}
```

[0147] The first step (i.e., using the procedure shown in **FIG. 4** and described herein above) is for the heuristic annotation generator to guess candidate annotations (step **402**) and to insert them into the program (step **404**). For this example, an annotated computer program **122** that results is shown in Table 3.

TABLE 3

Computer Program of Table 2 Annotated with Candidate Set

```
void main() {
    int x = 5;
    int y = timestwo(x);
    //@ assert y >= 0;
}
//@ requires n >= 0;              // candidate annotation 1
//@ requires n < 0;               // candidate annotation 2
//@ ensures \result >= 0;         // candidate annotation 3
//@ ensures \result < 0;          // candidate annotation 4
int timestwo(int n) {
    return 2*n;
}
```

[0148] The static checker, as described above, would process this program as follows. First, it would generate verification conditions VC_main and VC_timestwo (step **406**) for the two modules respectively:

[0149] VC_main is given by:

$$x=5\rightarrow x\geqq 0\Lambda x<0\Lambda(y\geqq 0\Lambda y<0\rightarrow y\geqq 0).$$

[0150] VC_timestwo is given by:

$$n\geqq 0\Lambda n<0\Lambda\ \text{result}=2*n\rightarrow\ \text{result}\ \geqq 0\Lambda\ \text{result}\ <0.$$

[0151] In the foregoing expressions "→" means IMPLIES, "Λ" means AND, and "Λ" binds more strongly than "→." Each verification condition is composed of a number of individual fragments separated from one another by conjunctions or disjunctions. For example, "result=2*n" is a fragment of VC_timestwo.

[0152] These verification conditions are passed to the theorem prover, whereupon the theorem prover will refute VC_main on account of the fragment "x<0" which comes from candidate annotation **2**. Such a refutation is presented as a warning **408**. At this stage, VC_timestwo is valid.

[0153] Because a warning of the second kind is issued and a candidate annotation is mentioned (step **410**), a counter example corresponding to the warning is mapped into an annotation modification (step **412**). A counter example to VC_main is, in this case, a simple formula, x=5.

[0154] Then, the refuted candidate annotation **2** is removed from the program (step **414**), yielding the first intermediate modified program **124** shown in Table 4:

TABLE 4

First Intermediate Modified Computer Program

```
    void main() {
        int x = 5;
        int y = timestwo(x);
        //@ assert y >= 0;
    }
    //@ requires n >= 0;           // candidate annotation 1
    //@ ensures \result >= 0;      // candidate annotation 3
    //@ ensures \result < 0;       // candidate annotation 4
    int timestwo(int n) {
        return 2*n;
    }
```

[0155] Now, the two verification conditions are generated for the first intermediate modified program by applying the program checking tool to it (step **406**):

[0156] VC_main is now given by:

$$x=5\rightarrow x\geqq 0\land(y\geqq 0\land y<0\rightarrow y\geqq 0).$$

[0157] VC_timestwo is now given by:

$$n\geqq 0\land \text{result}=2*n\rightarrow \text{result}\geqq 0\land \text{result}<0.$$

[0158] These verification conditions are passed to the theorem prover. This time, VC_main is valid but VC_timestwo is not on account of the fragment "result<0" which comes from candidate annotation **4**. Thus a warning is issued and the corresponding counter example is mapped into an annotation modification (step **412**). In this case, the counter example is n≧0Λ result=2*n, which is the left hand side of VC_timestwo.

[0159] Consequently, the refuted candidate annotation **4** is removed from the program (step **414**), yielding the second intermediate modified computer program shown in Table 5:

TABLE 5

Second Intermediate Modified Computer Program

```
    void main() {
        int x = 5;
        int y = timestwo(x);
        //@ assert y >= 0;
    }
    //@ requires n >= 0;           // candidate annotation 1
    //@ ensures \result >= 0;      // candidate annotation 3
    int timestwo(int n) {
        return 2*n;
    }
```

[0160] The verification conditions are generated for the second intermediate modified computer program (step **506**):

[0161] VC_main is now given by:

$$x=5\rightarrow x\geqq 0\land(y\geqq 0\rightarrow y\geqq 0).$$

[0162] VC_timestwo is now given by:

$$n\geqq 0\land(\text{result}=2*n\rightarrow \text{result}\geqq 0).$$

[0163] These two verification conditions are passed to the theorem prover, which finds both of them to be valid and issues no new warnings that mention candidate annotations. Hence, finally, the annotation inference system outputs a modified computer program **120** that contains just candidate annotations **1** and **3**.

Example 2

Using Guarded Verification Conditions to Avoid Modifying the Computer Program

[0164] The method according to the present invention employs an improved version of the static checker that avoids regeneration of the verification condition each time that a candidate annotation is refuted and removed from the program. This improved version of the static checker operates as follows, as shown in **FIG. 5**.

[0165] After the annotation assistant has generated a set of candidate annotations and inserted them into the program, a guarded verification condition (GVC) is generated for the annotated computer program (step **505**):

[0166] Using the program used in Example 1, and shown in Table 2 hereinabove, two guarded verification conditions are created, denoted GVC_main and GVC_timestwo, respectively.

[0167] GVC_main is given by:

$$x=5\rightarrow(G\mathbf{1}\rightarrow x\geqq 0)\land(G\mathbf{2}\rightarrow x<0)\land((G\mathbf{3}\rightarrow y\geqq 0)\land(G\mathbf{4}\rightarrow y<0)\rightarrow y\geqq 0).$$

[0168] GVC_timestwo is given by:

$$(G\mathbf{1}\rightarrow n\geqq 0)\land(G\mathbf{2}\rightarrow n<0)\land \quad \text{result}=2*n\rightarrow(G\mathbf{3}\rightarrow \text{result}\geqq 0)\land(G\mathbf{4}\rightarrow \text{result}<0).$$

[0169] Each fragment of the guarded verification conditions is given a guard variable name, for example G**1**, G**2**. Each fragment corresponds to a single annotation. Subsequently, a set of initial truth assignments, denoted TAO and stored in the guard truth vector associated with the GVC, is generated for the guard variables:

TAO: G**1**=trueΛG**2**=trueΛG**3**=trueΛG**4**=true.

[0170] The theorem prover is then presented (at step **507**) with the following two formulas:

TAO→GVC_main; and

TAO→GVC_timestwo.

[0171] The theorem prover will refute the first of these formulas on account of the fragment "x<0" and produce a counter example which comes from candidate annotation **2**. The second formula is valid at this stage.

[0172] The truth assignments for the guard variables are updated (step **515**) to record that candidate annotation **2** has been refuted. Thus, G**2** is set to false. The updated truth assignments are denoted TA**1** to indicate that the values are those set on the first iteration:

TA**1**: G**1**=trueΛG**2**=falseΛG**3**=true Λ**3** G**4**=true.

[0173] Next, the theorem prover is presented with the following two formulas (step **507**):

TA**1**→GVC_main; and

TA**1**→GVC_timestwo.

[0174] This time, the first formula is valid and the second is not on account of the fragment "result<0" which comes from candidate annotation **4**. Accordingly, a counter example results, and the truth assignments for the guard variables are updated (step **515**) to record that candidate annotation **4** has been refuted. The truth assignments, denoted TA**2**, are thus:

TA**2**: G**1**=trueΛG**2**=falseΛG**3**=trueΛG**4**=false.

[0175] Again, the theorem prover is presented (step **507**) with the following two formulas:

$$TA2 \rightarrow GVC\_main; \text{ and}$$

$$TA2 \rightarrow GVC\_timestwo.$$

[0176] This time, both formulas are valid and no more warnings that mention a candidate annotation are generated. Thus, the annotation inference system removes from the program annotations whose guards are false, step **516**.

[0177] In each of the foregoing steps, simplifications can be applied to the GVC combined with the guard truth vector. Such simplifications are often called 'peep-hole-like optimizations' because they concentrate on the details of small portions. Examples of peephole optimizations include the following. For a GVC whose form is:

$$(G0 \rightarrow P_0) \Lambda (G1 \rightarrow P_1)$$

[0178] wherein P_0 and P_1 are postconditions, if **G0** is true and **G1** is false, a first peephole optimization is to rewrite the GVC as:

$$(\text{True} \rightarrow P_0) \Lambda (\text{False} \rightarrow P_1).$$

[0179] In which case, each of the fragments may be further replaced, in another peephole optimization:

$$(P_0) \Lambda (\text{True}).$$

[0180] This expression can, in turn, be further simplified to P_0 because any expression and-ed with "true" is simply the expression itself.

[0181] Finally, the annotation inference module outputs a modified version of the program **(120)** that contains just candidate annotations **1** and **3**.

[0182] As shown, the annotation inference module generates the verification conditions (GVC_main and GVC_timestwo) only once, saving the time that would otherwise be required to re-parse and re-type-check the computer program and re-generate verification conditions.

Example 3

Comparison of Timings Using Guarded Verification Conditions Without Modifying the Computer Program With Prior Method

[0183] The ideas of the present invention have been implemented in the annotation assistant, Houdini, specifically, the version Houdini **2.0**, which infers annotations for ESC/Java. Houdini consists of three components. Two of these are components of ESC/Java: a verification condition generator, which has been modified to produce guarded verification conditions, and a theorem prover, Simplify. The third component is a driver program that implements the annotation inference logic.

[0184] Given a Java program, Houdini first generates an initial set of annotations and uses ESC/Java to produce guarded verification conditions for every procedure (method or constructor) of the program. The obtained guarded verification conditions are stored as text files on disk. The driver program contains two modules: the coordinator and the server. The coordinator remotely starts a fixed number of server processes and performs scheduling of verification tasks among them. Given a procedure name, a server loads its verification condition from disk, applies dropGuards and various peephole-like optimizations as described hereinabove in Example 2, to it. The server then sends the obtained formula to a local copy of the theorem prover Simplify, and forwards the results of the verification back to the coordinator. The coordinator and servers communicate using sockets.

[0185] Communication overhead of this annotation assistant is insignificant, and the running time is dominated by the theorem proving component. The coordinator process idles about 90% of the time waiting for replies from the servers. Each server process, in turn, spends between 5% to 10% of its time preparing the verification condition for the theorem prover; Simplify takes the rest of the time.

[0186] Principal experiments have been conducted on four input programs:

[0187] Java2Html, a 500-line program that turns Java programs into color-coded HTML pages (Compaq Systems Research Center, available from research.compaq.com/SRC/software);

[0188] WebSampler, a 2,000-line program that performs statistical samplings of trace files generated by the web crawler Mercator (see, Heydon, A., and Najork, M. A., "A scalable, extensible web crawler, "*World Wide Web,* 2(4):219-229, (1999));

[0189] PachyClient, the 11,000-line graphical user interface of the web-based email program Pachyderm (Compaq Systems Research Center, (1997), available from research.compaq.com/SRC/pachyderm); and

[0190] Cobalt, a proprietary 36,000-line program.

[0191] Table 6 shows some statistics about these programs.

TABLE 6

The benchmark programs used for the performance numbers in Example 3.

	lines			Annotations		Warnings	
benchmark	of code	classes	routines	candidate	valid	ESC/Java	Houdini
Java2Html	558	5	32	398	86	70	11
WebSampler	1875	14	127	6252	5061	252	41
PachyClient	10928	57	653	33062	6076	1325	525
Cobalt	36152	173	1157	28363	9246	3978	649

[0192] Table 6 also shows how many candidate annotations were guessed for these programs and how many of the candidate annotations were valid. The difference between these two numbers is how many annotations Houdini refuted. For WebSampler, most of the candidate annotations remain, which is because the files included a lot of code that was not reachable from the given program entry points. Accordingly, as can be seen by comparing PachyClient and Cobalt, the number of candidate annotations is not just a function of program size but also of the particular declarations in the program.

[0193] The last two columns in Table 6 show how many warnings ESC/Java and Houdini produced on these programs. That is, they show how many warnings ESC/Java produced on the programs without any annotations and also with the valid annotations inferred by Houdini. As can be seen, the number of warnings is decreased significantly by employing an annotation inference system.

[0194] Table 7 shows some performance numbers on the four benchmark programs. The computations detailed in Table 7 were performed on a single processor.

TABLE 7

Houdini analysis performance.

	Generate		New Houdini	
benchmark	candidate annotations	Original Houdini	Generate GVC	
Java2Html	0:0:08	0:5:44	0:0:20	0:3:17
WebSampler	0:0:12	1:11:32	0:2:03	0:28:39
PachyClient	0:0:42	35:21:25	0:49:44	6:41:47
Cobalt	0:4:41	60:00:00(*)	0:18:24	11:47:34

[0195] All times in Table 7 are in hours:minutes:seconds. Numbers marked with (*) in Table 7 are approximations. The first number column shows that the time to generate the candidate annotations is insignificant compared to the rest of the computation. The next column shows the time required by the unoptimized Houdini to refute the invalid candidate annotations.

[0196] The last two columns of Table 7 pertain to Houdini runs that use the optimizations described hereinabove, in particular using the fastest-first work ordering heuristic (but not the no-overlap heuristic). The first of these columns shows the time required to generate the guarded verification conditions from the annotated program. The proportionately long time taken for PachyClient is due to the large number of annotations that were input to the program. The last column shows the time taken from entering the iterative loop to exiting it. This time is dominated by the time taken to refute candidate annotations. Time taken for additional steps, such as for the program updater to remove from the annotated computer program those annotations that have been refuted, are not shown in Table 7. It can be seen from Table 7, that the methods of the present invention give rise to a significant improvement in performance over previous comparable methods.

Example 4

Heuristic Generation of Annotations

[0197]

TABLE 8

Example program for which annotations are generated.
Line numbers as shown in the left hand column.

```
 0 int a[100];
 1 int b[50];
 2 int m;
 3 int n;
 4 int x  := 0;
 5 int y  := 0;
 6 while (x < 100) {
 7    m  := a[x];
 8    n  := b[y];
 9    x  := x+2;
10    y  := y+1;
11 }
```

[0198] Running a conventional program checking tool on the program shown in Table 3 produces the following warning:

Line 8: possible array index out of bounds.

[0199] An annotation inference system according to the method of the present invention first guesses many annotations and then lets the underlying tool refute them. For the program shown in Table 8, the annotation inference module may guess the annotations shown in Table 9.

TABLE 9

Candidate set of annotations heuristically derived for the program in Table 8

0 <= x
0 <= y
x <= 50
y <= 50
x <= 100
y <= 100
y <= x
x even
x odd
y even
y odd

[0200] The rationale behind the annotations in the candidate set is as follows. The number zero (0) is important to many programs, especially as a likely lower bound to an array index variable such as x or y. The integer fifty (50) appears in the program text as the length of array "b," and hence may also be an important value. Similarly, the integer one hundred (100) appears in the program text as the length of array "a." Other guesses utilize the variables in various possible conditions.

[0201] After repeatedly calling a program checking tool to refute these annotations, the set of annotations that remains is shown in Table 10.

TABLE 10

Set of annotations remaining in the program of Table 8 after iteratively calling a program checking tool

0 <= x
0 <= y
x <= 100
y <= 100
y <= x
x even

[0202] Note that the loop invariant "y<=50" has been refuted, despite the fact that it is true in all executions. The reason for this is that there is no explicit link between the variable y, the array incrementing variable x and the bounds of execution of the loop. Had the heuristic annotation generator also guessed a condition like "y+y=x," then both "y<=50" and "y+y=x" would have remained unrefuted.

[0203] Note also that a condition like "x=0Vx>=2" is a loop invariant, but it was not guessed by the annotation assistant in this example because of the overall simplicity of the annotations.

Alternate Embodiments

[0204] The present invention can be implemented as a computer program product that includes a computer program mechanism embedded in a computer readable storage medium. For instance, the computer program product could contain the program modules shown in **FIG. 1**. These program modules may be stored on a CD-ROM, magnetic disk storage product, or any other computer readable data or program storage product. The software modules in the computer program product may also be distributed electronically, via the Internet or otherwise, by transmission of a computer data signal (in which the software modules are embedded) on a carrier wave.

[0205] While the present invention has been described with reference to a few specific embodiments, the description is illustrative of the invention and is not to be construed as limiting the invention. Various modifications may occur to those skilled in the art without departing from the true spirit and scope of the invention as defined by the appended claims.

What is claimed is:

1. A method of annotating a computer program with a least one unrefuted annotation, comprising:

inserting a set of candidate annotations into the computer program to create an annotated computer program;

generating at least one guarded verification condition from said annotated computer program wherein said guarded verification condition comprises a set of guards wherein each guard in said set of guards corresponds to an annotation in said set of candidate annotations and wherein an initial truth value of each of said guards is set to true;

applying a theorem prover to the at least one guarded verification condition, to produce one or more counter examples;

for each of said counter examples that indicates that there is an inconsistency between the computer program and at least one annotation in said set of candidate annotations:

updating the truth value of each guard that corresponds to said at least one

annotation;

repeating said applying and said updating until said theorem prover produces no counter examples that indicates that there is an inconsistency between the computer program and an annotation in said set of annotations; and

modifying the computer program so as to remove every annotation whose truth value has been updated thereby creating a modified computer program that contains at least one unrefuted annotation.

2. The method of claim 1 wherein said set of candidate annotations is derived by employing a heuristic analysis of the computer program.

3. The method of claim 2 wherein said set of candidate annotations comprises a candidate invariant for a variable f.

4. The method of claim 3 wherein said candidate invariant comprises an expression that includes a comparison operator.

5. The method of claim 4 wherein said expression includes an operand selected from the group consisting of: a variable declared earlier in a same class of the computer program; any one of the constants −1, 0, 1; and a constant dimension in an array allocation expression in the computer program.

6. The method of claim 2 wherein said set of candidate annotations comprises a predicate selected from the group consisting of: a precondition and a postcondition.

7. The method of claim 1 additionally comprising, after said modifying, applying a program checking tool to said modified computer program to present one or more warning messages to a user.

8. The method of claim 7 wherein said program checking tool is an extended static checker.

9. The method of claim 1 wherein said theorem prover is contained within an extended static checker.

10. The method of claim 1 wherein said generating additionally comprises:

converting said annotated computer program into an intermediate form; and

processing said intermediate form by a verification condition generator to produce said at least one guarded verification condition.

11. The method of claim 10 wherein said processing utilizes a guarded weakest precondition operator.

12. The method of claim 10 wherein said processing additionally comprises:

producing at least one verification condition and subsequently transforming said at least one verification condition into said at least one guarded verification condition.

13. The method of claim 12 wherein said producing utilizes a weakest precondition operator.

14. The method of claim 1 wherein at least one of said counter examples that indicates that there is an inconsistency between the computer program and at least one annotation in said set of candidate annotations, corresponds to a warning message.

15. A computer program product for use in conjunction with a computer system, the computer program product comprising a computer readable storage medium and a computer program mechanism embedded therein, the computer program mechanism comprising:

a set of instructions for inserting a set of candidate annotations into a computer program;

a verification condition generator for generating at least one guarded verification condition from said annotated computer program wherein said guarded verification condition comprises a set of guards wherein each guard in said set of guards corresponds to an annotation in said set of candidate annotations and wherein an initial truth value of each of said guards is set to true;

a theorem prover for producing, from the at least one guarded verification condition, one or more counter examples;

a guard truth vector updater for updating the truth value of each guard that corresponds to an annotation that, according to at least one of said counter examples, is inconsistent with the computer program;

control instructions for iteratively applying the theorem prover and guard truth vector updater until said theorem prover produces no counter examples that indicates that there is an inconsistency between the computer program and an annotation in said set of annotations; and

instructions for modifying the computer program so as to remove every annotation whose truth value has been updated thereby creating a modified computer program that contains at least one unrefuted annotation.

16. The computer program product of claim 15 further comprising a heuristic annotation generator for heuristically deriving said candidate set of annotations from the computer program.

17. The computer program product of claim 16 wherein said set of candidate annotations comprises a candidate invariant for a variable f.

18. The computer program product of claim 17 wherein said candidate invariant comprises an expression that includes a comparison operator.

19. The computer program product of claim 18 wherein said expression includes an operand selected from the group consisting of: a variable declared earlier in a same class of the computer program; any one of the constants −1, 0, 1; and a constant dimension in an array allocation expression in the computer program.

20. The computer program product of claim 16 wherein said set of candidate annotations comprises a predicate selected from the group consisting of: a precondition and a postcondition.

21. The computer program product of claim 15 additionally comprising instructions for applying a program checking tool to said modified computer program to present one or more warning messages to a user.

22. The computer program product of claim 21 wherein said program checking tool is an extended static checker.

23. The computer program product of claim 15 wherein said theorem prover is contained within an extended static checker.

24. The computer program product of claim 15 wherein said verification condition generator additionally comprises:

instructions for converting said annotated computer program into an intermediate form; and

instructions for processing said intermediate form to produce said at least one guarded verification condition.

25. The computer program product of claim 24 wherein said instructions for processing utilize a guarded weakest precondition operator.

26. The computer program product of claim 24 wherein said instructions for processing additionally comprise:

instructions for producing at least one verification condition and instructions for subsequently transforming said at least one verification condition into said at least one guarded verification condition.

27. The computer program product of claim 26 wherein said instructions for producing utilize a weakest precondition operator.

28. The computer program product of claim 15 wherein at least one of said counter examples that indicates that there is an inconsistency between the computer program and at least one annotation in said set of candidate annotations, corresponds to a warning message.

29. A system for annotating a computer program with at least one unrefuted annotation, the system comprising:

at least one memory, at least one processor and at least one user interface, all of which are connected to one another by at least one bus;

wherein said at least one processor is configured to annotate the computer program with at least one unrefuted annotation; and

wherein said at least one processor executes instructions to:

insert a set of candidate annotations into the computer program;

generate at least one guarded verification condition from said annotated computer program wherein said guarded verification condition comprises a set of guards wherein each guard in said set of guards corresponds to an annotation in said set of candidate annotations and wherein an initial truth value of each of said guards is set to true;

apply a theorem prover to produce, from the at least one guarded verification condition, one or more counter examples;

update the guard truth vector so that the truth value of each guard that corresponds to an annotation that corresponds to at least one of said counter examples, is inconsistent with the computer program;

iteratively apply the theorem prover and invoke the guard truth vector updater until said theorem prover produces no counter examples that indicates that there is an inconsistency between the computer program and an annotation in said set of candidate annotations; and modify the computer program so as to remove every annotation whose truth value has been updated thereby creating a modified computer program that contains at least one unrefuted annotation.

30. The system of claim 29 wherein said at least one processor further executes instructions for heuristically deriving said candidate set of annotations from the computer program.

31. The system of claim 30 wherein said set of candidate annotations comprises a candidate invariant for a variable f.

32. The system of claim 31 wherein said candidate invariant comprises an expression that includes a comparison operator.

33. The system of claim 32 wherein said expression includes an operand selected from the group consisting of: a variable in a same class of the computer program; any one of the constants −1, 0, 1; and a constant dimension in an array allocation expression in the computer program.

34. The system of claim 30 wherein said set of candidate annotations comprises a predicate selected from the group consisting of: a precondition and a postcondition.

35. The system of claim 29 wherein said at least one processor further executes instructions for applying a program checking tool to said modified computer program to present one or more warning messages to a user.

36. The system of claim 35 wherein said program checking tool is an extended static checker.

37. The system of claim 29 wherein said theorem prover is contained within an extended static checker.

38. The system of claim 29 wherein said at least one processor additionally executes:

instructions for converting said annotated computer program into an intermediate form; and

instructions for processing said intermediate form to produce said at least one guarded verification condition.

39. The system of claim 38 wherein said instructions for processing utilize a guarded weakest precondition operator.

40. The system of claim 38 wherein said instructions for processing additionally comprise:

instructions for producing at least one verification condition and instructions for subsequently transforming said at least one verification condition into said at least one guarded verification condition.

41. The system of claim **40** wherein said instructions for producing utilize a weakest precondition operator.

42. The system of claim 29 wherein at least one of said counter examples that indicates that there is an inconsistency between the computer program and at least one annotation in said set of candidate annotations, corresponds to a warning message.

* * * * *