Expression and Localization of Object Invariants

by

T. Benjamin Self

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 17, 2000

Copyright 2000 T. Benjamin Self. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and
distribute publicly paper and electronic copies of this thesis
and to grant others the right to do so.

Author_________
Department of Electrical Engineering and Computer Science
May 17, 1998

Certified by_________
Daniel Jackson
Thesis Supervisor

Accepted by_________
Arthur C. Smith
Chairman, Department Committee on Graduate Theses
Expression and Localization of Object Invariants

by

T. Benjamin Self

Submitted to the Department of Electrical Engineering and Computer Science on May 17, 2000 in Partial Fulfillment of the Requirements for the Degrees of Bachelor of Science in Computer Science and Engineering and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

One of the most difficult things in determining correctness of programs is the checking of cross-object invariants. Often, it is very simple to check whether or not an invariant is upheld inside of one object, but it is no trivial task to check these same constrains when then need to be upheld across objects. In this project, I have developed a tool, Hawk, that will help a user determine whether or not an invariant is upheld by isolating the section of Java code that is needed for the user to look at in order to determine whether or not the invariant is upheld. It does this by first building an object model of the compiled Java code, and the allowing the user to input the invariant based upon this object model.

Thesis Supervisor: Daniel Jackson
Title: Ross Professor of Software Technology
Acknowledgments

I would like to thank my thesis advisor, Daniel Jackson, for his knowledge throughout this process.

I would also like to thank ever-evolving room 511 crew, Eric Bothwell, Albert Lee, Allison Waingold, István Molnar, Mandana Vaziri, and Coco, for their help around the office.

Thanks to Rebecca Breazeale for her friendship and love which made MIT a much better place to be, and my friends from old 400, who made my stay here enjoyable.

And of course, my parents, Tom and Phyllis Self, who made my stay at MIT, and this thesis, possible. They were the ones who instilled in me the drive that allowed me to come here.

Thomas Benjamin Self,

May 17, 2000
Contents

1. Introduction
   1.1 Objective
   1.2 Testing of Programs
   1.3 Object Modeling
   1.4 Organization of this Report

2 Object Modeling Tools
   2.1 Alloy
   2.2 Alcoa
   2.3 Womble

3 Hawk
   3.1 Hawk’s Functionality
   3.2 Using Hawk
   3.3 Technical Organization

4 CTAS Case Study
   4.1 CTAS
   4.2 Aircraft Module

5 Evaluation of Hawk
   5.1 Successes
   5.2 Deficiencies
   5.3 Future Directions for Work
   5.4 Conclusion
Chapter 1

Introduction

1.1 Objective

There are two goals that I hoped to achieve with this project. First, I wanted to create a tool that helps programmers check the correctness of their code. It would accomplish this by helping the user isolate the sections of Java code that should be checked in order to determine whether or not the assumptions made from the program design are actually upheld in the code. Secondly, I wanted to evaluate the usefulness of this tool to determine whether or not it would actually be useful in an industry environment.

1.2 Testing of Programs

As computers get faster and faster, programs are able to get more and more complex. This complexity causes the ability to check the correctness of a particular piece of code to become more and more difficult. Many times, it is simply not possible to glass box test every possible case, as it was when most programs were less complex and contained fewer modules. It is important during these times, however, to be able to still guarantee the correctness of the code, especially in software where code failure could cost human lives. In the past, there have been many examples where poor software testing led to the death of several people, such as the Therac-25 disaster[2].

Quality assurance testing is one of the most important aspects of good software engineering. Many researchers estimate that the time needed to determine correctness of a program is often as high as 40% of the entire process. Most companies employ QA testers at a ratio of one to every two developers. However, most of the tools that are currently used only guarantee correctness based on test cases that are specifically determined by these testers. For example, the user will input several thousand possible test cases and watch the program to make sure it performs to specifications without any
problems during these cases. However, even if there are no problems found with the code, there is still no guarantee that the program will work for all possible inputs, only that it works correctly for the inputs provided

This inability to manually guarantee correctness has created the need for better error checking tools. The tool that I have created, called Hawk, first builds an object model of the Java code analyzed. It then allows the user to enter the specific invariants that need to be upheld during the life of the program, and helps the user determine whether or not it is possible to break these invariants based on an analysis of the current bytecode. Although it does not check correctness by itself, it helps the user by isolating the section of source code the user needs to look at in order to determine whether or not the invariant it upheld. Therefore, its output is a listing of functions in the current program module that the user needs to look at to determine whether or not an invariant is upheld in the code. Before this project, no such tool existed. Test suites only existed which could guarantee with a percent of certainty that certain specific invariants were upheld, based on the specific test cases they were given.

1.3 Object Modeling

An object model of a program is a representation of the abstract state of the program. It can take the form of a graph, where the nodes represent sets of objects, and the arcs represent relationships (subsets or associations) between those objects, or the form of a textual description where the objects and their relationships are enumerated.

The code structure of an object oriented program can often be represented through a highly structured object model. This object model is a map of the program, helping the programmer to determine quickly and easily how the objects relate to each other.

While traditional design documentation, such as dependency diagrams and call graphs are based on the procedural structure of the code, the object model of a program focuses on its "architecture". This allows for an object model to represent things that are becoming increasingly more important as programs become increasingly dynamic, with many components being configured at runtime. An object model can represent all of these possible structures and can therefore help the programmer analyze many possible configurations of the system[1].

1.4 Organization of this Report

This thesis is organized into six chapters. This first chapter has explained some of the basic motivations for doing this research. The second chapter will provide a description of the technologies used to create Hawk. The third chapter will explain the usage and technical details of Hawk, and the fourth chapter will apply Hawk to two case studies to show its usefulness. Finally, the fifth chapter will discuss the successes and shortcomings of this research, and also discuss future directions for work.
Chapter 2

Object Modeling Tools

The technologies in Hawk were created by combing and augmenting the technologies in several other object model tools and technologies. The Alloy language is the object modeling language of choice for Hawk. The object modeling parsing abilities of Alcoa are used to parse and check the invariants provided by the user to Hawk, and the ability to create a graphical object model present in Womble is used as a basis to form the textual object model present in Hawk[5].

2.1 Alloy

Alloy is an object modeling language. It is a lightweight, precise and flexible notation for object modeling developed by Professor Daniel Jackson at MIT. It “attempts to combine the practicality of UML’s static structure notation with the rigor of Z and the to be expressive enough for most object modeling problems while remaining amenable to automatic analysis.” “It also has simple set-based semantics, and a type system that, by treating scalars as singleton sets, allows relations and functions to be treated uniformly, and sidesteps the problem of undefined expressions.”[1]

Alloy is best explained through the use of an example.

```
model Family {
  domain {person, name}
  state {
    partition Man, Woman : static Person
    Married : Person
    Parents : Person -> static Person
    Siblings : Person -> Person
    Wife (-husband) : Man ? Woman ?
    Name : Person -> Name ?
  }
  def siblings {
    all a, b | a in b.siblings <-> (a.parents = b.parents)
  }
}
```
inv Basics {
    all p | some p.wife <-> p in Man & Married
    no p | p.wife / in p.siblings
    all p | (sole p.parents & Man) && (sole p.parents & Woman)
    no p | p in p.+parents
    all p,q | p.name = q.name -> no (p.parents & q.parents)
}

op Marry {
    m not in Married && w not in Married
    m.wife' = w
    all p | Men - m | p.wife' = p.wife
    all p | p.name' = p.name
    all p | p.parents' = p.parents
    Person' = Person
}

assert HusbandsWife {
    all p : Married & Woman | p.husband.wife = p
}
}

In an Alloy textual model, the object model is broken down into several different schemas: domain, state, def (definitions), op (operations), inv (invariants), and asserts. In the domain and state schemas, the different types of sets of objects are defined and their relationships (hierarchical and associations) are also enumerated. The domain schema lists the top layer objects—in this case person and name. The state schema then describes the relationships between these objects. Take, for example, the first line in the state schema:

\[\text{partition Man, Woman : static Person}\]

This says that the Person domain is made up of two subsets, Man and Woman. These partitions are static—that is, once an object is instantiated in one of these subsets, it cannot move to the other. This is an example of how a hierarchy can be formed using Alloy.

The second line:

\[\text{Married: Person}\]

says that Married is a subset of Person. This is an example of a classification.

The third line:

\[\text{parents : Person -> static Person}\]

describes a relationship between two objects. The relationship parents is an association from one person to another person. In addition, this relationship is right-static—that is, once a person object in the program is created and given a set of parents, that set can not change. The rest of the syntax for Alloy, including descriptions for the other schemas, can be found in [3].
It may not, however, be easily apparent how this object modeling language can be applied to a piece of source code. The following example will hopefully illuminate this tie.

Consider the following block of Java source code:

```java
public class Person
{
    private String Name;
    private int Age;
    private Woman Mother;
    private Man Father;
    protected boolean Married;
    private Vector Siblings;
    private Person Children[];

    public Person()
    {
        Name = "";
    }

    public void Birth(Woman m, Man f, String n)
    {
        Name = n;
        Mother = m;
        Father = f;
        Age = 0;
    }

    public void setName(String t)
    {
        Name = t;
    }

    public void setAge(int i)
    {
        Age = i;
    }

    public void ChangeName(String t)
    {
        setName(t);
    }

    public class Man extends Person
    {
        private Woman Wife;

        public Man()
        {
        }

        public void Marry(Woman w)
        {
            Married = true;
            Wife = w;
        }
    }
```
public void Divorce()
{
    Married = false;
    Wife=null;
}
}

public class Woman extends Person
{
    private Man Husband;

    public void Marry(Man m)
    {
        Married = true;
        Husband=m;
    }

    public void Divorce()
    {
        Married = false;
        Husband=null;
    }
}

Although this piece of code would not be represented with the object model described before, a simpler version of that object model could be used. This object model, created by Hawk, will be shown in Chapter 3.

2.2 Alcoa

Alloy was created in order to make it easy to do automatic analysis of object models. The tool created in order to do this analysis is Alcoa. Alcoa can perform a deep semantic analysis of any text object model that can be created with Alloy[5]. It can check the consistency of constraints (invariants), generate sample configurations which fit these constraints, simulate executions of operations described in an alloy model, and check to make sure that these operations uphold the constraints.

Although Alcoa has a lot of functionality, very little of it was used when creating Hawk. The Alloy object model parser was removed from the Alcoa source code and added into Hawk. This includes the syntax and type checker built into Alcoa. This functionality will be described further in Chapter Four.

2.3 Womble

Womble is a tool, created by Allison Waingold, that can extract the underlying object model from Java bytecode[6]. It uses a quick analysis of the bytecode to create this object model, whose nodes are the Java classes, and whose edges are either subclasses or associations. It can be run in several different modes, and creates a graphical version of the relations selected, with labels and mutability constraints. Its output is a file that can easily be translated to a postscript file.
An example of Womble’s output, when applied to the previous example, can be found on the right side of figure 1. To the left is a graphical representation of the textual Alloy model found earlier in this chapter. As you can see, they are very similar, however, the basic differences in the code and in the object model are easily apparent.

![Diagram of Womble's output](image)

**Figure 1**

This is an extremely useful tool for programmer. It is simple and easy to use, and can be used to quickly determine whether or not the structure of a program follows what the programmer expected it to be. The object model of the program created by Womble can then be compared to the object model of the program created by the software designer, and then this can be used as a first determination of whether or not the program was written correctly.
Chapter 3

Hawk

Hawk is a tool created in order to help a programmer check the correctness of his code. Hawk combines and augments functionality from Alcoa and Womble in order to create a tool which will allow a programmer to parse Java bytecode, and then determine whether or not the code lives up to certain specification. It isolates the sections of source code that should be analyzed to determine whether or not the invariants in the code are upheld.

3.1 Hawk’s Functionality

Hawk’s inputs are the source code for a program, and the invariants that the programmer is concerned with. The output is a listing of object method names that must be looked at by the user to determine whether or not the invariant is upheld by the program. The following convoluted, but illustrative example can help to show the extended functionality of Hawk.

Let’s consider again the Family example from the previous chapter. Using the source code presented in section 2.1, when Hawk is run with the invariant

all a | some a.Person_name   (which means: all people objects must have some name)

The output is as follows:

To Determine whether the invariant:
   all a | some a.Person_Name
is upheld, the following relations are important to look at:
   Person.Person_Name
These relations are modified in the following functions:
Relation: Person.Person_Name
Functions:
   Person.Person_<init>
   Person.Person_Birth
   Person.Person_setName
There are several classes of output that are displayed here. In the first four lines of the output, Hawk gives a brief description of what task it just performed. It then tells the user what relations are important to pay attention to (in this case Person.Person_Name), and then tells the users which functions must be looked at in order to determine whether or not the invariant is upheld. Although they are not clearly displayed in the current version, these functions are organized into two groups. The first group (Person.Person_<init>, Person.Person_Birth, Person.Person_setName) are first degree functions, meaning that they directly affect the truth of the invariant. It is in these functions that the relation the invariant depends may be modified. The second group of functions listed (Man.Man_<init>, Woman.Woman_<init>, Person.Person_ChangeName) are the second degree group of functions, which are put there to help the user determine the context in which the first degree functions are called. These functions are called second degree because they directly call the first degree functions. If there were third degree functions, they would be displayed after the second degree functions.

3.2 Using Hawk

The user interface of Hawk is fairly simple (see Figure 2). Once the program is run, the first thing that must be done is that the user must select the directory which the class files are located in. The user selects a directory by using the open command from the file menu.

![Figure 2]
Once the user has opened the directory with the class files in it, Hawk parses all the files that end in class (*.class) located in that directory and then builds the object model based on these Java classes. It compiles a list of main Java objects, and also of all the relations between them. Because there might be two different relations that have the same name (a “Man” object might have a relation “name”, and a “Dog” object might have a relation “name”), Hawk makes these names unique by adding the class name to it (i.e. these would become “Person_name” and “Dog_name”). This is done because the object model parser located in Alcoa would not accept two identically named relations with different objects. This is very common programming practice, however, so adding the class name to the relation name increases the usability of Hawk.

After this is done, Hawk prints out the Alloy model, giving it a dummy name of “Analyzed”. The object model that Hawk would generate for the previous Family example follows:

```alloy
model Analyzed {
  domain {Person, Man, Woman, String, Vector}
  state {
    Person_Name: Person -> String !
    Person_Mother: Person -> Woman ?
    Person_Father: Person -> Man ?
    Person_Siblings: Person -> static Vector ?
    Person_Children: Person -> static Person
    Man_Wife: Man -> Woman ?
    Woman_Husband: Woman -> Man ?
  }
}
```

Once this is finished, the user can create any invariant based on the model that the user would like to check in the code. This invariant is put in the text entry box in the upper right of the screen, and then the “Find” button is pressed. The invariant is then parsed, and if there is an error with the invariant, this is displayed in the output area. If there is no syntax or type errors, the output, as described in the previous section, is then displayed in the lower right hand text display area.

### 3.3 Technical Organization

The parsing of the class files and the creation of the object model is completed by adding a layer of abstraction to the Womble v1.4 code. There was an API created that allows the files to be parsed, and the objects placed in a certain structure. This API allows internal access to the ModelMaker object inside of Womble. There are then accessor functions that are added into the API that will get the information needed about each type of object that is needed in order to create the API.
The model is then created by parsing all the *.class files in the selected directory, and building up a vector of all the classes that are used by the files. When an object is added to the vector, its parents are also recorded in an attempt to build up hierarchical associations. A second vector is created which contains all the associations between objects.

Finally, all of these collections of object and associations are parsed together into several strings that make up the object model. The list of objects is concatenated together to form the body of the domain schema, and the associations are parsed, put together, and built up in order to form the body of the state schema. Note that not only are the types of objects and their associations listed in the state schema, but number and mutability are also discovered and displayed.

Next, the user enters the invariant to be checked by Hawk. This invariant is read in as a String, and then added as a new schema, an Inv schema (invariant) to the object model. This complete object model is then passed to the model class inside of Alcoa. The model is then parsed by Alcoa and compiled in order to populate all the schemas needed to get out the information needed by Hawk. Once this is complete, a visitor is created which will visit all of the objects of type FormulaSchema inside Alcoa, which will produce a list of all of the relations referred to inside of the invariant. For example, when considering the invariant:

\[ \text{some } a \mid a.\text{Man}_\text{Wife}.\text{Woman}_\text{Husband} = a \]

the program will output both:

\[
\begin{align*}
\text{Man}_\text{.Man}_\text{Wife} \\
\text{Woman}_\text{.Woman}_\text{Husband}
\end{align*}
\]

Finally, in the third step, the program attempts to isolate all of the sections of source code where this invariant is upheld. It does this by reparsing the byte code, this time looking for functions in which these objects are modified. Specifically, there are only certain instructions in the Java bytecode which can modify a relation. Hawk looks for these bytecode instructions, and the notes what functions they are called from. So, in this case, it looks for the relation Man.Man_Wife first and builds a list of all the functions where this relation is modified. After building this list and outputting this, it then searches for all the functions that call these modifying functions, and lists them. This is done by looking for calls from the bytecode which reference this function. Note that it is simply looking for times that the function might be called, not that it actually is called in every case. This process is then repeated a third level deep. Hawk then repeats this entire process for all the other relations that resulted from the Alcoa object model parsing. This is how the functions that are displayed to the user are found.
Chapter 4

CTAS Case Study

In this chapter, I analyze an example of Hawk applied to problems with CTAS, NASA’s air traffic control system. I begin by giving a brief overview of CTAS, and then give a few specific examples of how this program can help in finding bugs in large programs. While this thesis is not an exercise to find bugs in large programs, I hope to show that in certain cases, the functionality present in Hawk can help a QA engineer to more quickly isolate the sections of source code that are important to look at.

4.1 CTAS

The Center TRACON Automation System (CTAS) is a large scale object oriented program which contains many different complex invariants. This application contains thousands of lines of code and many different objects which are dynamically created and can be organized into many different possible configurations. Therefore, it is extremely difficult to guarantee correctness of the code in this complex system.

CTAS is a suite of tools designed at the National Aeronautics and Space Association (NASA) which attempts to help air traffic controllers manage the complex and stressful task of organizing the finite airspace over airports in the United States. These tools are designed to increase the rate at which airplanes can land at airports, which, in most cases, is the limiting factor in air traffic control. It takes into account many different factors about an airplane, and attempts to predict its aircraft trajectory as much as 40 minutes in advance[4]. A crash in the system forces the air traffic controllers to manage the airplane using only a pencil and paper method.

4.2 Aircraft Package

First, I am going to apply Hawk to Daniel Jackson’s Java version of the aircraft package of CTAS, which is located in an aircraft directory. This is a fairly small module,
containing only about 8 classes and 400 lines of code. The object model generated by Hawk is:

```alloy
defined Analyzed {
  domain { Aircraft, AircraftId, AircraftTable, DuplicateException, Flight_plan_st_obj, CTAS_flight_info_st_obj, Cm_ac_st_obj, Meter_fix_id_st_obj, String }
  state {
    Aircraft_id: Aircraft -> static AircraftId !
    Aircraft_fp: Aircraft -> static Flight_plan_st_obj !
    Aircraft_fl: Aircraft -> static CTAS_flight_info_st_obj !
    Aircraft_state: Aircraft -> Cm_ac_st_obj ?
    Aircraft_c: Aircraft -> Meter_fix_id_st_obj ?
    AircraftId_id: AircraftId -> String
    AircraftTable_aircrafts: AircraftTable ? -> AircraftId
    AircraftTable_aircrafts: AircraftTable -> Aircraft
  }
}
```

Let’s say that a user wanted to make sure every aircraft had an ID and a flight plan. A possible Alloy invariant to express this would be:

```alloy
all a: Aircraft | some a.Aircraft_id && some a.Aircraft_fp
```

After imputing this invariant, the program quickly reports the following:

To Determine whether the invariant:

```alloy
all a: Aircraft | some a.Aircraft_id && some a.Aircraft_fp
```

is upheld, the following relations are important to look at:

- Aircraft.Aircraft_id
- Aircraft.Aircraft_fp

These relations are modified in the following functions:

- Relation: Aircraft.Aircraft_id
- Functions:
  - Aircraft.Aircraft_<init>
- Relation: Aircraft.Aircraft_fp
- Functions:
  - Aircraft.Aircraft_<init>

Therefore, without ever looking at the code directly, a user can tell that the only place where these two variables are modified is in the constructor functions. The user can tell that they are never modified or erased. The user can also determine that since they are set in the constructor, then all Aircraft objects must have one (except if there are conditional statements – see the final chapter for more explanation of this case). Therefore, the user can safely assume that this invariant is upheld without ever having looked at the code.

Now, let’s say that the programmer wanted to see something a bit more complicated. One possible invariant that must be upheld is the following:

```alloy
all a, s | (a! = s.Aircraft_id) -> a.Aircraft_id.AircraftId_id != s.AircraftId_id
```
This says that for every Aircraft (a), and Aircraftid (s), if aircraft_id is not equal to s, then the string inside a’s aircraft_id (a.Aircraft_id.Aircraftld_id) cannot equal the string inside s (s.AircraftId_id).

Running this invariant in Hawk produces the following:

To Determine whether the invariant:
all a,s | (a!=s.Aircraft_id) -> a.Aircraftid.Aircraftld_id != s.AircraftId_id
is upheld, the following relations are important to look at:
Aircraft.Aircraft_id
Aircraftld.Aircraftld_id
These relations are modified in the following functions:
Relation: Aircraft.Aircraft_id
Functions:
Aircraft.Aircraft_<init>
Relation: Aircraftld.Aircraftld_id
Functions:
Aircraftld.Aircraftld_<init>
AircraftTable.AircraftTable_get

What this shows is much more interesting, because it provides more information. This shows the user that Aircraft_id is only modified in Aircraft’s constructor (which we knew from the previous example). In addition, it also shows the programmer that Aircraftld.Aircraft_id is only changed inside Aircraftld’s constructor, and that this constructor is also called from AircraftTable_get. This is where the example is most interesting. A normal user should then look inside the AircraftTable function to see why the function get was reported:

```
public static Aircraft get(String acname) {
    Aircraftld ac = new Aircraftld(acname);
    return actable.lookup(ac);
}
```

This is a very surprising example. In order to look up a certain ID string in the AircraftTable, it creates an Aircraftld object, and then looks for that. This is a surprising occurrence – why create a dummy Aircraftld object just to look it up? In fact, when looking at this function, the invariant IS broken every time an aircraft is looked up in the AircraftTable. This simple example shows the usefulness of Hawk. It took only a few minutes to determine whether or not an invariant can be broken. Of course, to determine if this case matters, it is important to look in the other modules and determine what modules call AircraftTable.gets.
Chapter 5

Evaluating Hawk

Although a lot of progress has been made in producing Hawk, there are still many things that can be done to make the tool better. In this section, I am going to expound on the successes of Hawk and also discuss its failures and ways that it would be possible to improve it.

5.1 Successes

In order to evaluate Hawk, it is important to first understand the rationale behind each of the design decisions made, and then it is important to decide whether or not they were in fact the correct decisions.

One design decision that is commonly questioned, is that when a variable is set, why does Hawk not report where that variable comes from? It seems to make sense that Hawk should also research where a variable comes from, since that is also important in determining whether or not an invariant is upheld. In short, this is a difficult problem. In addition, it is quite often that a variable is set based on a value that is passed into a function, which is determined at runtime. Therefore, although Hawk will report what functions will call the modifying function, it does not directly attempt to discover the source of the data.

Another thing important to look at in order to determine whether or not Hawk was a success is how much code does Hawk report to the user when run over an average relation. The following statistics were generated based on analyzing four packages, with 45 relations of CTAS. All of these correspond to fields that are of object type (i.e. not primitives like int, double, etc.).
<table>
<thead>
<tr>
<th>Packages</th>
<th>Average Number of First Degree Functions</th>
<th>Standard Deviation of First Degree Functions</th>
<th>Average Number of Second Degree Functions</th>
<th>Standard Deviation of Second Degree Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>All Packages</td>
<td>1.05</td>
<td>0.21</td>
<td>1.66</td>
<td>1.71</td>
</tr>
<tr>
<td>Aircraft Package</td>
<td>1.29</td>
<td>0.49</td>
<td>0.14</td>
<td>0.38</td>
</tr>
<tr>
<td>Alviq Package</td>
<td>1</td>
<td>0</td>
<td>0.60</td>
<td>0.52</td>
</tr>
<tr>
<td>Message Language Package</td>
<td>1</td>
<td>0</td>
<td>3.2</td>
<td>1.32</td>
</tr>
<tr>
<td>Message Package</td>
<td>1</td>
<td>0</td>
<td>0.29</td>
<td>0.49</td>
</tr>
</tbody>
</table>

These statistics show that the amount of information reported to the user was far from overwhelming. In most cases, there were one or two first degree functions reported, and one or two second degree functions reported. This little amount of information helps the user to quickly determine which parts of the code are relevant and which parts do not affect what the user is concerned with.

These packages examined are four of the five present in the Java version of CTAS. Many of them only have one first degree function because they are static relations which are only set in the constructor method of the object, and never changed anywhere else. In addition, many of these functions have very few second order functions because they are called from other packages in CTAS. Therefore, the small number of second degree functions may appear deceiving.

As you can see from the previous examples, Hawk has the potential to be a very useful tool to software developers. It can manage very large bodies of code quickly and effectively, and build object models out of this code in order to simplify it. Hawk’s runtime is linear in the size of the analyzed program’s size, therefore, it can handle large, complex object models. In addition, it can very quickly scan and determine where relations are modified, not only outputting these functions, but also outputting the functions that called those functions.

In addition to being useful for its intended purpose, Hawk also has several unintended byproducts that make it extremely useful to programmers. First, the program is useful for building a textual alloy model of the code, which can be compared and contrasted to the software designer’s version of the object model. This simple comparison can help the programmer determine which sections of the code need work. Although most of this work is done through Womble, Womble cannot output a textual Alloy object model that can be analyzed using Alcoa. This functionality in itself is extremely useful to a programmer.
Secondly, Hawk is helpful to a programmer intending to reverse engineer a program. By helping the user determine the call trees of a specific Java program, Hawk can sharply decrease the time needed to correctly understand a program. Although this is simply a side result of the program, it is extremely useful when attempting to check the affects of a particular bug. For example, if it is discovered that there is a bug in the function fooO(), Hawk can be used to determine everywhere fooO() is called, and the programmer can therefore quickly determine how widespread the affects of the bug are. Hawk is an extremely useful tool when trying to come up to speed on a program very quickly. It can enable the programmer to understand a lot of information about a program very quickly, and also allow the programmer to manipulate different invariants and learn a lot about the functionality and the structure of the code.

Thirdly, a useful feature of Hawk is that it works only with the object code of the program. Therefore people who do not have access to the source code but who are dependent on its functionality can determine the correctness of the code before building software that is dependent on it.

Finally, Hawk is extremely useful in doing what it was intended to do – help the programmer determine the correctness of important invariants. Although it does not check the correctness itself, it can greatly decrease the amount of time needed to determine the correctness of an invariant. In the examples I have shown, the usefulness of this tool is easily apparent.

5.2 Deficiences

One shortcoming of Hawk is that sometimes it does not produce enough information to the user. For example, if there was an object of type Man (subset of Person), with a relationship “wife”, and there is a function that that changes the relationship wife of a Person object (Person.Person_wife), Hawk will not recognize this as a change to a Man.Man_wife relation. Therefore, if there was an invariant that is based on this relation, Hawk might not report any functions to check when in fact the relation is not upheld. Although this is a current problem, this is not technically beyond the scope of the program, and it is quite possible that in the future, Hawk could be expanded so that it could account for this case.

Similarly, Hawk cannot resolve dynamic method calls. If there is a method that has previously been determined to modify a relation in the invariant, and that method belongs to a subclass of an object then if that method is called by calling the method of the superclass, Hawk will not realize that the method call could be a method call of the subclass. For example, if the method foo() in the Man object modifies the relation determined to be important (i.e. foo() is a first degree function), and there is some other function, bar(), which calls Person.foo() without first casting that Person object to a Man, then bar() will not be reported as a second degree function because it does not explicitly call the function Man.foo().
Hawk also does not deal well with functions with conditional cases. Although it may appear that a relation is set in an \texttt{init} function, it could be the case that this relation is only set if certain conditions are met. Therefore, it is extremely important that the user double check before any assumptions are made about the content of a block of code.

5.3 Future Directions for Work

There are several characteristics of Hawk that could be updated or fixed in order to increase the functionality of the tool. Although none of these faults cripple the effectiveness of Hawk, with their addition, it could become an incredibly useful tool.

First, the user interface of the program could be polished and updated. Although everything needed for functionality is there, it could be organized in a much better, more intuitive way. In addition, the output of the program is in a very rough form. If a relation is mentioned twice in an invariant, it will be searched for twice, and the program will do double the amount of work. In addition, the call tree of functions is not clearly specified in the output. Although all of the information is resident inside of the program, it is not displayed to the user at this point.

Secondly, there is currently a lack of support for Java base classes in the program. This is especially evident when considering functions such as vector and array. The program will currently only notice when the vector is completely set to a new vector, not when the vector itself changes. This is a large shortcoming that could be avoided with the following addition: functionality needs to be added that will parse the classfiles.zip file and determine the set of mutators inside of each of the major Java classes. These mutators could be stored in some sort of database, and then they could be searched for just like a function is searched for when looking up a call tree. Therefore, the main functionality is built, and all that is needed is the classfiles.zip parser. However, this is a more difficult problem than it seems. When a user wants to know whether or not a Vector class is modified, the user really wants to know when the member variable elts is modified inside the Vector class. This is more difficult to search for because it is not something the program can infer – it has to be told this before it is run. Therefore, each of the main Java classes have to be annotated to know this data (such as modifying vector really means modifying elts, and modifying HashTable really means modifying the data values elements of the data structure, not the keys of the HashTable).

5.4 Conclusions

Determining the correctness of software invariants is a very difficult problem that few software developers can do effectively. For the average programmer, Hawk could be a useful tool in order to help check the correctness of their Java code. However, in order to become an extremely useful tool, the functionality and interface need to be improved. Hopefully, this thesis will create some interest in this topic, and the development of Hawk can continue, so that it can become the tool that it was intended to be.
Bibliography


