Concurrent Software Verification with Explicit Transfer of Permission

Theodore S. Norvell
Electrical and Computer Engineering
Memorial University of Newfoundland
Email: theo@mun.ca

***Abstract*—Concurrent software is difficult to reason about and impossible to verify by testing. Yet much software written today is concurrent. We therefore need practical mechanisms to automatically verify concurrent software. I propose a way of annotating concurrent programs based on fractional permissions, dynamic frames, and explicit transfer of permissions. An annotated program may be automatically verified by translating to a sequential intermediate verification language such as the Boogie IVL and then verifying a set of sequential fragments.**

CONTENTS

I. INTRODUCTION

The concurrent programming language Harpo [1], [2] is an object-oriented, concurrent programming language intended for high-performance, embedded applications. Programs are nondeterministic because threads execute asynchronously. Memory is potentially shared between threads.

There are several sources for error in Harpo programs. These fit into two categories.

- Static errors including syntax errors, use of undeclared identifiers, incorrect number of arguments for a method call, type errors
- Dynamic errors. These fit into two categories
  - There are several sources of undefined behaviour, i.e., situations where the language definition does not specify what an action means. Examples of undefined behaviour include array's being indexed out of bounds, arithmetic overflow, two threads accessing the same memory location at the same time where one access is a write.
  - Failure to ensure the desired output or the desired course of action. In this case the program behaves in a way defined by the language definition, but it does not behave in the way intended by the designer of the software.

Static errors can be prevented statically, using standard compilation techniques.

The aim of the present work is to create a method whereby all dynamic errors can also be found. Our intention is that

A common approach to finding and diagnosing dynamic errors is testing. While software testing is very useful for many reasons, it is not suitable for ensuring that software is free of all dynamic errors. First any test only really gives us information about one possible input. Information about that one input might lead to some reasonable assumptions about similar inputs, but the highly nonlinear nature of software means that reasonable assumptions are often not true. Furthermore because the language is inherently nondeterministic, a passed test with

one input does not imply that the program's behaviour for that input will always be acceptable. What testing is useful for is for gathering examples of behaviours that lead to dynamic errors; such examples can be of great help to the engineer fixing the program. I will say more about testing in the concluding section.

The essence of our approach is to pass permissions between thread and other threads or between objects and threads. And to require that all memory accesses to potentially shared locations be made only by threads that have sufficient permission.

II. Permissions

Talking sticks are used in some Indigenous North American societies to ensure that only one person speak at a time. The stick is passed from person to person and only the person who holds the stick is allowed, by convention, to speak. In our system, each location is associated with a permission and only the thread that holds that permission may write to the location. In order to prevent read/write collisions, we also require that a thread must have permission to read as well. Permissions may also reside with objects. To allow multiple threads to read a location at the same time, we allow a thread to pass only a fraction of its permission to another thread. We represent permissions as real numbers in the interval $[0, 1]$. The conventions of the language will ensure that the sum of all permissions existing on a single location will always sum to 1 or less. Thus if a thread has permission 1 on a location, it is safe to write to the location and if it has permission that is greater than 0, it is safe to read from the location. This fractional permission system is similar to that used in Chalice [3].

At run time, a Harpo program consists of a set of locations, a set of objects, a set of arrays, and a set of threads. Each location is capable of holding any member some primitive type. Locations may be fields of objects, items of an array, or top-level locations. Some locations are local to a thread, but these can not be shared, so we won't worry about them. For each thread or object t there is a mutable map p_t that maps permissions to real numbers in the interval $[0, 1]$. As a global invariant we ensure, for all locations l, $(\sum_t p_t(l)) \leq 1$.

(**class** *Ex1*
  obj i : **int** := 0 ;
  claim i ;
  invariant $permissionOn(i) = 1 \wedge i \geq 0$
  ⋮
class)

Algorithm 1: A class with a claim

(**class** *Ex2*
  obj j : **int** := 0
  (**thread claim** j
    ⋮
  thread)
class)

Algorithm 2: A thread with a claim

III. Details

A. Initial distribution

As part of the start up process for a Harpo program, all objects are created. This static nature of Harpo is an artifact of its intended implementation in hardware and its intended application in safety critical systems. The language could be extended to allow dynamic creation (and finalization) of objects. In that case permission on existing locations would need to be donated by the creating thread.

Each class declares the amount of permission its objects will initially have on locations. For example the class shown in Algorithm 1 declares that each object of the class initially has full permission (1.0) on its i field.

The invariant of the class says that the object maintains full permission on the field.

Similarly threads may claim permission at start up by a declaration as shown in Algorithm 2 . This example shows that

Threads and objects may claim less than full permission. For example, **claim** $k@0.5$ will claim half permission on the location.

B. Forking

Threads can create child thread though parallel composition and parallel looping. Each child thread

can claim some of parent's permission. At the end of a parallel command, all permission held by the child threads is returned to the parent. For example if the parent holds permission to all items of an array, a parallel loop can be written as follows

$$
\begin{array}{l}
(\textbf{co}\ i : 100\ \textbf{do} \\
\quad \textbf{claim}\ a(i) \\
\quad S \\
\textbf{co})
\end{array}
$$

Here, each child thread obtains permission from its parent on one item of the array.

C. Conditional critical sections

In Harpo, threads can attain exclusive access by locking an object. The command is

$$(\textbf{with}\ o\ \textbf{when}\ G\ \textbf{do}\ S\ \textbf{with})$$

where o is an object reference G is an optional guard and S is a command. Essentially, this is Hoare's conditional critical section [4]. At most one thread can lock any object; a thread that tries to lock an already locked object will wait until it can gain exclusive access to the object. At the start of s, the object's invariant may be assumed. For example, if the class of the object o is Ex1, the designer may assume that prior to the execution of S, o has full permission on $o.i$ and also that $o.i \geq 0$. During the execution of S, the permissions of the thread are added to the permissions of o (and any other objects locked by syntactically surrounding code) and so if S reads or writes to $o.i$, that is allowed. It is an obligation on the designer to ensure that the object's invariant true when the object is locked. So if S were

$$o.i := o.i - 1$$

it would be an error. On the other hand code, if S were

$$(\textbf{if}\ o.i > 1\ \textbf{then}\ o.i := o.i - 1\ \textbf{if})$$

that would be fine.

Object invariants must be self-supporting in that they must guarantee that any change made by a thread that has not locked the object will not cause the invariant to change from true to false. Looking again at Ex1, if the invariant were simply $i \geq 0$, that would not be self-supporting since a thread that has not locked the object could change i if it had permission 1 on the location. The idea of self-supporting invariants is a form of dynamic framing [5].

$$
\begin{array}{l}
(\textbf{class}\ \textit{Ex3} \\
\quad \textbf{obj}\ k : \textbf{int} := 0 \\
\quad \textbf{obj}\ a : \textbf{array}\ 100\ \textbf{of int} \\
\quad \textbf{claim}\ k, \{i \in \{k, ..100\} \cdot a(i)\} \\
\quad \textbf{invariant}\ \mathit{permissionOn}(k) = 1 \\
\quad \wedge\ 0 \leq k \leq 100 \\
\quad \wedge\ (\forall i \in \{k, ..100\} \cdot \mathit{permissionOn}(a(i)) = 1) \\
\quad \vdots \\
\textbf{class})
\end{array}
$$

Algorithm 3: A class with a dynamic frame

Threads can use conditional critical sections to transfer permission from and to objects. Consider an object with class Ex3 in Algorithm 3. The claim here claims all locations in the array.

A thread can obtain permission on a segment of the array with

$$
\begin{array}{l}
(\textbf{with}\ o\ \textbf{when}\ o.k \geq 10\ \textbf{do} \\
\quad \textbf{takes}\ \{i \in \{o.k, ..o.k + 10\} \cdot o.a(i)\} \\
\quad p := o.k \\
\quad o.k := o.k + 10 \\
\textbf{with})
\end{array}
$$

The **takes** annotation indicates that permission on 10 locations should be transferred from the object to the thread at the start of the conditional critical section. Since the object's invariant is assumed, at the start, it can be inferred that the object has the permission. Similarly permission can be given by the thread to the object at the end of a conditional critical section. For example a thread that has full permission on items. For example a thread could to the following, provided it started with full permission on items $\{p, ..p + 10\}$ of array $o.a$:

$$
\begin{array}{l}
(\textbf{with}\ o\ \textbf{when}\ o.k = p + 10\ \textbf{do} \\
\quad o.k := p \\
\quad \textbf{gives}\ \{i \in \{p, ..p + 10\} \cdot o.a(i)\} \\
\textbf{with})
\end{array}
$$

(**class** *Server*
  public method $start($
    in $p : \mathbf{int}, \mathbf{in}\ q : \mathbf{int}, \mathbf{out}\ s : \mathbf{real})$
  pre $0 \leq p \leq q \leq 100$
  takes $\{i \in \{p, q\} \cdot a(i)@0.5\}$
  post $s' = (\Sigma i \in \{p, q\} \cdot a(i))$
  gives $\{i \in \{p, q\} \cdot a(i)@0.5\}$
  ⋮
class)

Algorithm 4: A class with a method declaration

D. Rendezvous

While conditional critical sections provide a low-level mechanism for inter-thread communication and coordination, rendezvous provide a higher-level mechanism.

Each class can declare methods. Methods are annotated by pre- and postconditions. Clients must ensure preconditions are true prior to the call. Servers must ensure that the postconditions are true after the call. It is also possible to transfer permission from the client to the server at the start of the call (takes) and from the client to the server (gives). For example a class might declare a method as shown in Algorithm 4.

The server takes 50% permission on some of the locations of an array, sums the members of the array and then returns the permission back to the client thread. The client must have sufficient permission to start and is responsible for ensuring the precondition is true at the time of the rendezvous.

E. Other features

Some other features supported include ghost fields, assert commands, assume commands, and loop invariants.

IV. Conclusions and Further Work

In order to verify code using the annotations, we can translate to the Boogie IVL [6]. Although this translation is not yet automated, the method of translation has been worked out and tested manually. This work is reported in the thesis of Yousefi Ghalehjoogh [7] and will be detailed in a future publication.

In order to allow full functional specification and verification the language needs to be extended to include features such as functions and predicates as in Dafny [8].

Future work includes automation of the translation. Experimentation with the current system of annotation and with more features.

References

[1] T. S. Norvell, M. A. A. Tuhin, X. Li, and D. Zhang, "HARPO/L: A language for hardware/software codesign." in *Newfoundland Electrical and Computer Engineering Conference (NECEC)*, 2008.

[2] T. S. Norvell, "A grainless semantics for the HARPO/L language," in *Canadian Electrical and Computer Engineering Conference*, 2009.

[3] K. R. M. Leino, P. Müller, and J. Smans, "Verification of concurrent programs with Chalice," in *Foundations of Security Analysis and Design V*, ser. LNCS, vol. 5705, 2009.

[4] C. A. R. Hoare, "Toward a theory of parallel programming," in *Operating Systems Techniques*, C. A. R. Hoare and R. H. Perrott, Eds. Academic Press, 1972.

[5] I. T. Kassios, "Dynamic frames: Support for framing, dependencies and sharing without restrictions," in *FM 2006: Formal Methods*, ser. LNCS, vol. 4085, 2006.

[6] K. R. M. Leino, "This is Boogie 2," Microsoft Research, Tech. Rep., 2008, draft. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=147643

[7] F. Yousefi Ghalehjoogh, "Verification of the harpo language," Master's thesis, Memorial University, 2014.

[8] K. R. M. Leino, "Developing verified programs with Dafny," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 1488–1490. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2487050