Automatic Correction to Safety Violations in Programs

Muhammad Umar Janjua Alan Mycroft
Computer Laboratory, Cambridge University, CB3 0FD, UK
{umar.janjua, alan.mycroft }@cl.cam.ac.uk

Abstract

Our goal is an automatic, compile-time and incremental technique to compute corrections to safety property violations in a program. For a program P containing a set of violating traces T with respect to a given safety property, our method incrementally transforms P into a new correct program P_c such that P_c no longer exhibits the *same* behaviour as T. While making these series of *Correcting Transformations (CT)*, we ensure that first, previous correct behaviours in the erroneous program *are preserved* in the corrected program as well, and second, *no new* error traces are introduced with respect to the given property.

In this paper, we address those safety property violations resulting from incorrect interleavings of threads in a program. We analyse program computation trees and insert thread blocking primitives in the program in such a way that only erroneous program paths are pruned.

Categories and Subject Descriptors CR-number [*Programming*]: Languages

General Terms Languages, Algorithms, Concurrency

Keywords automatic, program correction, safety property, lock, multithreading

1. Introduction

When a programmer encounters an error in a program, the next important tasks are: firstly, to localise the cause of the error, and secondly, to make a correction to the erroneous program. Current automatic techniques only provide a certain level of help in this regard. Generally, when the program is erroneous, these techniques can report the presence of error during program execution (testing), or suggest that the program cannot be proved to be correct (theorem proving) or provide long counterexample traces (model-checking). These techniques do not provide information on the causes of the error or the ways of correcting the erroneous programs.

For example, in the case of model checking, the reported counterexamples do not exactly pinpoint the cause of the error. The execution path modelled by the counterexample is only known to be erroneous once the model checker makes a transition (which is often the last transition) to an erroneous state. Before that, there is a possibility that it might not be an erroneous execution. Hence, the exact cause of the error remains elusive with in the entire counterexample trace. Also, these counterexamples may be generated in the more highly abstracted models of the program. Because of abstraction, it becomes quite a tedious task for the programmer to relate the transitions in counterexamples to the statements in the actual source code.

Even after time-consuming model checking, it is still then left to the programmer alone to identify a possible cause and make a manual correction in the program. We argue that this very process of manually fixing or correcting erroneous program is itself cumbersome and error prone. For a non-trivial sequential program with nested conditional and loop statements, it becomes generally difficult to keep all possible program paths in view. This situation becomes annoyingly intractable for multithreaded programs and may result in the following undesirable consequences.

- Such a restricted view may lead to introduction of new errors in the program while trying to correct the previous errors.
- Even if no new errors are introduced, there still lurks a possibility that previous correct execution behaviours might have changed as a result of new corrections.

In this paper, we present a new automatic way of correcting error traces in the programs. One can envisage a development cycle where a programmer first develops a program implementation of a specification, second, verifies the implementation with a model checker, and third, if a counterexample is reported, a correction tool synthesizes a correction to the error, which can be later incorporated into the program on programmer's approval. Such a framework would reduce debugging time, avoid introduction of new errors, and facilitate the understanding the program in general.

These automatic corrections can also be viewed as another form of error explanation [4] to the error traces. Error explanation is a way of explaining error traces by pinpointing potential statements as the cause of the error. Our view is that, rather than highlighting a few "suspicious" statements to the programmer as a potential cause of the error, instead, it will be far more useful to iteratively provide possible corrections to the these error traces.

The contributions of this paper are :

1. Proposing an incremental scheme for automatic correction of safety violations.
2. Determining corrections/fixes while ensuring
   - No new program errors are introduced with respect to the safety property.
   - Previously correct program behaviours are preserved.

The remainder of this paper is structured as follows: Section 2 explores a simple motivating example to illustrate the technique. Section 3 provides background definitions. Section 4 describes correcting transformations. Section 5 gives a simple case study with a prototype tool. Section 6 discusses related work and Section 7 concludes and suggests further work.

Thread Verification '06 21 Aug 06, Seattle.

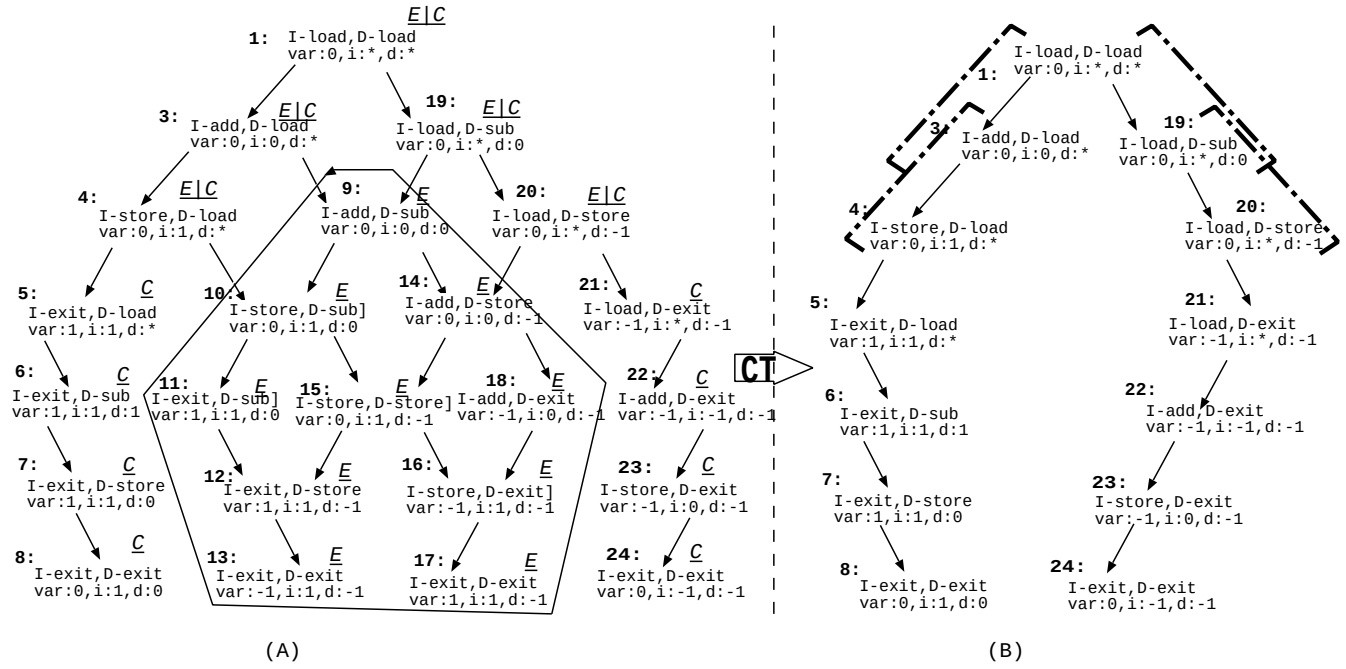

Figure 1. (A) A program computation tree containing both correct and error traces,w.r.t assert(var == 0). $\underline{E}$ is an Error node, $\underline{C}$ a Correct node, while $\underline{E/C}$ an E/C node. (B) contains only correct traces as a result of removing error traces. Brackets around two nodes indicate that these should be enclosed in an atomic block.

2. Motivating Example

Consider two concurrent threads I and D sharing a single processor, which increment and decrement a shared global variable var using thread-specific local temporary variables, `i` and `d`, respectively, as shown in the Figure 2.

```
var=0
void increment() { // for thread I
        i = load var
        i = add i, 1
        store i, var
        exit() }

void decrement() { // for thread D
        d = load var
        d = sub d, 1
        store d, var
        exit() }
```

Figure 2. An erroneous program with two concurrent threads I and D

Each instruction is atomic, and threads only interrupt in between these instructions. The initial value of var is zero. Suppose the programmer desires that following property holds: ***After both threads exit, its value must be zero***.

The complete computation tree of this program is shown in part (A) of Figure 1. State 1 in the Figure 1 says that either the thread I or D can execute different $load$ instruction of $increment$ and $decrement$ functions, respectively, when var has value 0, and i,d have undefined values. The execution of I in any state is shown by $\swarrow$, and that of D by $\searrow$.

The correct traces are $\{1, 3, 4, 5, 6, 7, 8\}$, $\{1, 19, 20, 21, 22, 23, 24\}$. All other traces lead to a violation of the specified property, because the var gets the value -1 or 1 in final states. Our goal is to preserve correct traces and remove error traces.

```
var=0
void increment() { // for thread I
⌊ i = load var
⟨i = add i, 1 ⌋ store i, var ⟩
exit() }

void decrement() { // for thread D
d = load var
d = sub d, 1
store d, var
exit() }
```

Figure 3. A semi-correct program (containing fewer error traces than in Figure 2)

Consider the error trace $\{1, 3, 4, 10, 11, 12, 13\}$. This violation occurs because the thread D interrupted thread I by loading var into d. This particular execution of D at state 4 leading to state 10 in Figure 1 is erroneous and undesirable. To restrict this execution of D, a correct solution could be to block D simultaneously with add in state 3 before the program reaches state 4, thereby only allowing I thread to be able to execute in state 4. Once the program goes beyond the state 4 to a "safe" state like 5, the D thread could be simultaneously enabled with $store$ again. This is logically equivalent to enclosing add and $store$ in an atomic section shown by $\langle$ and $\rangle$ in semi-correct program (contains fewer error traces) in Figure 3. Later, we show how to achieve this with thread blocking correction primitives as in section 4.3.

Similarly repeating the same procedure for error trace $\{1, 3, 9, 14, 15, 16, 19\}$ requires blocking thread D in state 1 atomically with $load$ and enabling it with add in state 3. This is shown by $\lfloor$

and ⌋. The program in Figure 3 is not fully correct because it still can exhibit the error trace $\{1, 19, 20, 14, 18, 16, 17\}$. Analysing the computation tree of this corrected program, we can incrementally correct remaining error traces as well. Note that these atomic sections overlap with each other, so these can be optimised into a single atomic section. Performing this optimisation would give a kind of correction which meets the programmer's intuition as well.

3. Definitions

3.1 Program Graph

We represent a program P as a union of control flow graphs of all procedures, assuming that there are only integer global variables, and no procedure calls. Each thread t executes a distinct procedure $proc_t$. Each thread exit $threadexit()$ call terminates the currently executing thread.

A control flow graph for each $proc_t$ is a directed graph $G_t = N_t, E_t, entry_t, exit_t$ with nodes N_t each containing an instruction, edges E_t representing the flow of control between nodes and $entry_t \in N_t$ and $exit_t \in N_t$ as the start and exit nodes. Then, $P = \bigcup_t G_t$.

3.2 Program State

Let S be a set of program states. A program state $s \in S$ comprises

- $\langle i_0, i_1, \cdots, i_{n-1} \rangle \in N_0 \times N_1 \times \cdots \times N_{n-1}$, a vector containing currently executable instructions for each of n threads.
- $\langle l_0, \cdots, l_t, \cdots, l_{n-1} \rangle$, where each $l_t \in Z$ represents the status of thread t. The condition $l_t \geq 1$ allows the execution of thread t, while $l_t \leq 0$ disallows the execution of thread t.
- V, is a store mapping shared global variable names to their current integer values, and also thread-local temporary variables to their current integer values.

Let $s_0 \in S$ be the initial program state.

3.3 Program Transition Relation

A Program Transition Relation is a non-deterministic function, $\sigma : S \rightarrow 2^S$; the non-determinism is used to represent the arbitrary interleaving of concurrent threads, while $\emptyset \in 2^S$ represents termination or deadlock.

3.4 Safety Labelling function

As in model-checking, we assume a predicate on states, $SL : S \rightarrow bool$ which indicates whether the state satisfied the user-safety property or not.

3.5 Program Trace or Path

A program trace T is a sequence of states $(s_0, s_1, \cdots, s_n)$ such that $\forall i, s_{i+1} \in \sigma(s_i)$. The set of all traces of P is $Traces(P)$, which depicts the total behaviour of the program.

However, for our purposes, rather than just considering a set of traces, it is more useful to form these traces into a tree as in the next section.

3.6 Program Computation Tree

A program computation tree is an unfolding of the program transition relation σ starting with the initial state $s_0 \in S$. Each computation tree node, ctn, contains a state $st(ctn)$, a single parent node $pt(ctn)$, and a set of possible j successor nodes $succ_j(ctn) \in \sigma(st(ctn))$. A root node is a ctn such that $st(ctn) = s_0$, while the leaf nodes have no successors.

A *Full computation tree* is a program computation tree where all possible successors of every node in the tree are included. A *Partial computation tree* is a program computation tree which is not a full computation tree.

Depending upon the given safety property, we classify computation tree nodes as

Error Node A ctn is an error node, if either its state violates the property, or all of its successor nodes' states violate the property. i.e $\neg SL(st(ctn)) \vee \forall j, \neg SL(st(succ_j(ctn)))$. In Figure 1, all those nodes which are enclosed in the polygon are error nodes.

Correct Node A ctn is a correct node, if it state satisfies the property, and all of its successor nodes' state satisfy the property, i.e $SL(st(ctn)) \wedge \forall j, SL(st(succ_j(ctn)))$. In Figure 1, the ctn 5,6,7,8,23,24 are correct nodes.

E/C Node A ctn is an E/C node, if at least one of its successor is an Error Node, and at least one of its successor is a Correct Node. In Figure 1, the ctn 3,4,1 are E/C nodes. Note that leaves cannot be E/C nodes.

3.6.1 Error Trace

An error trace is a path in the computation tree starting from the root node to the leaf such that the final node is an error node. In Figure 1, $\{1, 3, 9, 10, 11, 12, 13\}$ is an error trace.

3.6.2 MCP:Maximal Correct Prefix /MEP:Minimal Erroneous Postfix of Trace

A maximally correct prefix (MCP) of an error trace T is a prefix of T beginning with the root node to the $last$ E/C node in T. A maximal erroneous postfix (MEP) of the error trace T is a postfix of T starting with the *first* error node and ending with the *last* error node. In Figure 1, for error trace $\{1, 3, 9, 10, 11, 12, 13\}$, the MCP is $\{1, 3\}$ and MEP is $\{9, 10, 11, 12, 13\}$. Note that, our approach is to preserve MCP of the trace and eliminate MEP as a result of automatic correction.

3.6.3 Error Lattice

We define an error lattice in Figure 4 with abstract values $\{\perp$, *Correct*, *Error*, *E/C*$\}$ having order $\perp \sqsubseteq Correct \sqsubseteq E/C$ and $\perp \sqsubseteq Error \sqsubseteq E/C$, where $\top = E/C, \perp = bot$. The abstract value E/C shows that the node has both erroneous and correct behaviour.

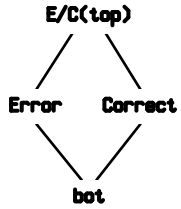

Figure 4. Error Lattice

3.7 Correction Primitives

Our focus is to correct errors which result from undesired interleavings of threads in the program. To control thread interleavings, languages and libraries provide synchronization primitives like monitors in Java and locks in the Posix thread library. Rather than using a particular language/library synchronization scheme, we use following simple but effective primitives and show how to prune off a maximal erroneous postfix of an erroneous trace.

A *block* operation on thread t is equivalent to $l_t = l_t - 1$ and an $unblock$ operation on thread t is equivalent to $l_t = l_t + 1$

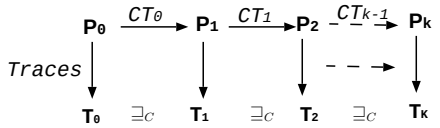

Figure 5. Correcting Transforms

BlockAll() blocks all threads except the currently executing one.

Block(t) only blocks the thread t.

UnBlockAll() unblocks all threads.

UnBlock(t) only unblocks the thread t.

UnBlockIf(t,cond) unblocks the thread t if condition $cond$ is true.

4. Correcting Transformation (CT)

The correcting transformation reduces the original erroneous program P_e to P_c such that $(\forall k \in Traces(P_c))erase(k) \in Traces(P_e)$, where $erase(k)$ is the trace obtained by removing correction primitives from trace k. When such a relation holds between program traces, we write it as $Traces(P_e) \sqsupseteq_c Traces(P_c)$ as shown in Figure 5. Our interest is not to completely re-engineer P_c. Instead, we insert correction primitives in a way that only MEP of the error trace is removed while preserving its MCP.

In Figure 6, consider the top level algorithm for automatic correction on `P` with safety labelling function `SL`. The function `findError` returns a partial computation tree rooted at `ctn` only if it contains an error trace. The function `CorrectTrans` attempts a correction. If a correction succeeds, `P` is transformed into `newP` using correction primitives and the entire process is repeated again. If a correction cannot be enacted, the algorithm terminates. The reason `CorrectTrans` could fail is that if by inserting correction primitives, the program could enter a deadlock state. (See Section 4.4).

```
procedure autoCorrect(var P )
while( true) {
  ctn = findError(P,SL);
  if(ctn == null)  break; // No error
  newP=CorrectTrans(ctn,P);
  if(newP == P)  break; //no change introduced
  else P=newP; // correction succeeded
}
```

Figure 6. Incremental Algorithm for automatic correction

4.1 Building and classifying Computation Tree nodes

We modify the traditional model-checking algorithm [5] to build and classify the computation tree nodes. In the forward phase, the algorithm explores unseen states in a depth-first manner and adds these to the computation tree with the default *bot* value. If an old state is visited again, or if there are no successor states from the current ctn, it is assigned $Correct$. While backtracking from the correct nodes, the parent computation tree nodes are marked as $Correct$.

When an error state is encountered, the current ctn is marked with $Error$ value. On backtracking from the error state, the error lattice value is propagated to the parent nodes by taking the least upper bound of all the successor nodes.

4.2 Determining Maximal Erroneous Postfix of Error Trace

There are two options:

1 One can fully explore the entire computation tree, and then propagate the error information upward by taking the least upper bound of all successor nodes. But for constraints of memory and speed,this is not practically feasible.

2 On the other hand, as in our case also, one can terminate state exploration after the first error state is reached. This only provides a partial computation tree which under-approximates the actual program behaviour. In the case where all of the ctn in the MEP of the error trace are fully explored, we have *exactly* determined MEP. However, this best case is not always achieveable. For partially explored ctn in the error trace, it is safe to assume a $Correct$ value for the unexplored successors of ctn. This is safe, because, our algorithm works incrementally. In the subsequent iteration, any error, if present, will be discovered.

4.3 Transformation

First, we explain the intuition behind our transformation. The analysis in Section 4.2 divides the error trace into its MEP and MCP. The goal of the transformation is to remove MEP while preserving MCP in the corrected program by inserting locking primitives. The transformation procedure centers around the E/C node. An execution of a thread in E/C node that leads to an error node is erroneous and undesirable. As an example, consider the execution of thread D from the state 4 to state 10 in 1. This thread D should be disallowed to execute before the program reaches state 4. However, once the program transits from state 4 to state 5, thread D should be allowed to execute again.

In Figure 7, we use the label e for an erroneous thread on the edge from E/C node to error node for the convenience of explanation. A correcting transformation should first block erroneous thread e *beforehand* in the parent state of E/C node, so that when program reaches state $st(ctn_{E/C})$, thread e is unable to execute in the E/C node, thereby eliminating MEP from this error trace. The Block(e) should be only executed by the active thread which transits the program from the $pt(ctn_{E/C})$ to $ctn_{E/C}$. In Figure 7, we use the label A for an active thread on the edge to E/C node from its parent for the convenience of explanation.

The blocked thread e should not remain blocked forever. Once the program is *about to* transit from E/C node to any of its correct successor nodes, the thread e should be "freed" to execute again in the successor nodes. This is achieved by inserting UnBlock(e) after each instruction of correct threads in E/C node.

The Block/UnBlock should execute for the particular valuation of store. To compare the stores, we have a predicate $comp : V_g \rightarrow V_{cur} \rightarrow bool$ which compares given store V_g with the current store V_{cur}.

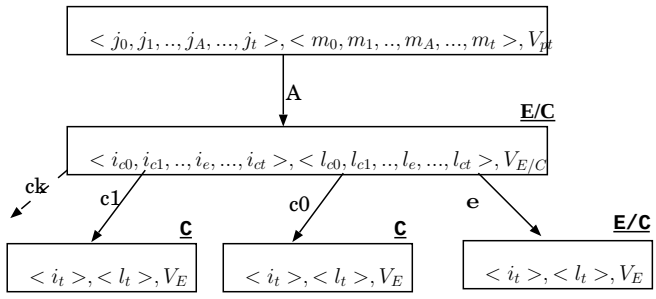

Figure 7. Transformation

In Figure 7, consider $ctn_{E/C}$ with its state $st(ctn_{E/C}) = \{< i_{c_0}, i_{c_1}, .., i_e, ..., i_{c_{n-1}} >,< l_{c_0}, l_{c_1}, .., l_e, ..., l_{c_{n-1}} >, V_{E/C}\}$. A thread e is a single erroneous thread that executes i_e in $st(ctn_{E/C})$ and transits the program to the error state $st(ctn_E)$. The threads

$c_k, \forall k, c_k \neq e$, are threads which lead to correct successor nodes when executed in $st(ctn_{E/C})$. Also consider $st(pt(ctn_{E/C})) = \{< j_0, .., j_A, .., j_{n-1} >, < m_0, ..., m_A, ..., m_{n-1} >, V_{pt}\}$ where A is an active thread that executes j_A in $st(pt(ctn_{E/C}))$ and transits program to $st(ctn_{E/C})$.

To block thread e atomically with j_A, instruction j_A is replaced with

BlockAll()

if($comp(V_{pt}, V_{cur})$) { j_A; Block(e);}

else j_A;

UnBlockAll()

To unblock thread e atomically with all $i_{c_k}, \forall k, c_k \neq e$, all instructions $i_{c_k}, \forall k, c_k \neq e$ in $st(ctn_{E/C})$ are replaced with

BlockAll()

if($comp(V_{E/C}, V_{cur})$) { i_{c_k}; UnBlock(e); }

else i_{c_k};

UnBlockAll()

4.4 Avoiding Deadlock

If erroneous thread e is also an active thread A, then performing transformation in Section 4.3 would result in a deadlock because thread A will block itself. Our analysis detects such a case and reports its inability to correct rather than producing a program which deadlocks.

4.5 Locking Policy

The transformation in the above section is based on the locking policy where thread $A, A \neq e$ is responsible for blocking thread e, and threads $c_k, \forall k, \wedge c_k \neq e$ are responsible for unblocking e. This strategy has the drawback that it fails to apply the correcting transformation when there is a risk of introducing deadlock. Another strategy could be to let the erroneous thread block itself, and assign responsibility to threads $c_k, \forall k, \wedge c_k \neq e$ to unblock e. Comparing different locking policies is included in the future work.

5. Example Case Study

```
int var=0;
my_pthread_t incr;
my_pthread_t decr;
void increment(void) {
       var=var+1;
       my_pthread_exit();
}
void decrement(void) {
       var=var-1;
       my_pthread_exit();
}
int main() {
       //creating incr and decr threads and waiting for exit
       ...
       ASSERT( var == 0);
}
```

Figure 8. C program containing violations to assertion (var==0)

We have implemented a prototype tool for a restricted subset of C language allowing only global variables, thread creation and exit, and disallowing function calls, stack and heap variables. The tool incrementally generates automatic corrections in the intermediate code representation [7] of the program. Consider a simple C program implementation in Figure 8 of the pseudocode in Figure 2 containing two threads, `incr, decr`. The unoptimised intermediate code in static single assignment form is shown in Figure 9. Here, `i.0, i.1` and `d.0,d.1` are thread-specific local temporary variables in `increment` and `decrement` function and each instruction is atomic.

```
void %increment() {
%i.0 = load int* %var
%i.1 = add int %i.0, 1
store int %i.1, int* %var
thread_exit()
}
void %decrement() {
%d.0 = load int* %var
%d.1 = sub int %d.0, 1
store int %d.1, int* %var
  thread_exit()
}
int %main() {  //code omitted
}
```

Figure 9. Initial erroneous intermediate program representation with race condition

For the purpose of understanding, the thread blocking primitives have been labelled on the LHS of the equality as "#–PEC" or "#–EC", where # gives the iteration number of correcting transformation, while PEC refers to the parent of EC node, and EC refers to EC node as depicted in the Figure 7.

```
void %increment() {
     %i.0 = load int* %var
 //atomically block decr with add instruction
    "1--PEC"=BlockAll()
     %i.1 = add int %i.0, 1
     "1--PEC"=Block(uint* %decr)
    "1--PEC"=UnBlockAll( )
 //atomically unblock decr with store instruction
     "1--EC"=BlockAll()
      store int %i.1, int* %var
     "1--EC"=UnBlock( uint* %decr )
     "1--EC"=UnBlockAll( )
      thread_exit( )
}
void %decrement() {
     %d.0 = load int* %var
     %d.1 = sub int %d.0, 1
      store int %d.1, int* %var
      thread_exit( )
}
```

Figure 10. After first correcting transformation

Figure 10 shows the result of applying first CT on the program in Figure 9. This correction eliminates the undesired interleaving of thread `decr` in between `add` and `store` instructions in the `increment` method only.

Figure 11 shows the result of applying second CT on the program in Figure 10. This correction further eliminates the undesired interleaving of thread `decr` in between `load` and `"1--PEC"=BlockAll()` instructions in the `increment` method. As a result of these two correcting transformations, the incr thread executes all three instructions atomically.

The result of applying third and fourth CT on the program in Figure 11 is collectively shown in fully corrected program in Figure 12. These subsequent corrections eliminate undesired interleavings of thread `incr` in between the instructions of the `decrement` method. As a result of these two correcting transformations, the `decr` thread executes all three instructions atomically.

```
void %increment() {
"2--PEC" =   BlockAll( )
%i.0 = load int* %var
"2--PEC" =   Block( uint* %decr )
"2--PEC" =   UnBlockAll( )
"2--EC" =   BlockAll( )
"1--PEC" =   BlockAll( )
"2--EC" =   UnBlock( uint* %decr )
"2--EC" =   UnBlockAll( )
%i.1 = add int %i.0, 1 ;
"1--PEC" =   Block( uint* %decr )
"1--PEC" =   UnBlockAll( )
"1--EC" =   BlockAll( )
store int %i.1, int* %var
"1--EC" =   UnBlock( uint* %decr )
"1--EC" =   UnBlockAll( )
 thread_exit( )
}
void %decrement() {
//same code as in Figure 10.
}
```

Figure 11. After second correcting transformation

6. Related Work

Griesmayer et. al. [2] have used game-theoretic setting to suggest local repairs to boolean programs(push-down models of programs). These repairs include corrections to incorrect variable assignments in the expressions.However, this strategy has not been shown to work for errors in concurrent programs. Earlier, there has been significant work on localising the error cause within counterexamples. Ball et. al [1] compare correct and incorrect execution traces in the program to identify culprit statements. Groce et. al [4] improve comparison by using distance-metrices. Choi and Zeller [3] use automated tests to pinpoint failure inducing thread interleavings. All these error explanation approaches are limited to merely highlighting suspicious statements/interleaving in the error traces and do not suggest corrections to the error, unlike our technique. The Autolocker tool [6] converts atomic sections specified in C programs to traditional lock-based code with the help of programmer annotations describing a link between shared data and protecting locks. Our approach does not require programmer to tediously specify these annotations, rather it automatically infers code instructions which should execute atomically to satisfy the desired safety property.

7. Conclusions and Further Work

This paper presents an incremental and compile-time technique to determine automatic corrections to safety property violations in restricted multithreaded programs. By appropriately inserting locking primitives in the program, only error traces resulting from undesired thread interleavings are removed while preserving previous correct behaviours. This is still a work in progress. Some important future directions include combining local automatic corrections into global corrections, comparing different locking schemes for granularity and determining suitable ways of comparing program state (global variable and stack variables) in corrected program. We would also like to observe how speed up techniques for model checking can improve our correction technique. We believe that writing and correcting concurrent programs are significantly difficult tasks, and in this context, to complement model-checking and error explanation techniques, our approach of proposing automatic corrections to errors will be very useful.

```
void %increment() {
"2--PEC" =   BlockAll( )
%i.0 = load int* %var
"2--PEC" =   Block( uint* %decr )
"2--PEC" =   UnBlockAll( )
"2--EC" =   BlockAll( )
"1--PEC" =   BlockAll( )
"2--EC" =   UnBlock( uint* %decr )
"2--EC" =   UnBlockAll( )
%i.1 = add int %i.0, 1
"1--PEC" =   Block( uint* %decr )
"1--PEC" =   UnBlockAll( )
"1--EC" =   BlockAll( )
store int %i.1, int* %var
"1--EC" =   UnBlock( uint* %decr )
"1--EC" =   UnBlockAll( )
   thread_exit( )
}
void %decrement() {
"3--PEC" =   BlockAll( )
%d.0 = load int* %var
"3--PEC" =   Block( uint* %incr )
"3--PEC" =   UnBlockAll( )
"3--EC" =   BlockAll( )
%d.1 = sub int %d.0, 1
"3--EC" =   UnBlock( uint* %incr )
"4--PEC" =   BlockAll( )
"3--EC" =   UnBlockAll( )
"4--PEC" =   Block( uint* %incr )
"4--PEC" =   UnBlockAll( )
"4--EC" =   BlockAll( )
store int %d.1, int* %var
"4--EC" =   UnBlock( uint* %incr )
"4--EC" =   UnBlockAll( )
   thread_exit( )
}
```

Figure 12. After fourth correcting transformation

Acknowledgments

The first author gratefully acknowledges funding from the Association of Commonwealth Universities and from Intel Research (Cambridge, UK) and useful comments from Hasan Amjad and Tom Harke.

References

[1] T. Ball and M. Naik and S. Rajamani. “From Symptom to Cause: Localizing Errors in Counterexample Traces”, POPL, 2003.

[2] Andreas Griesmayer, Roderick Bloem, and Byron Cook. “Repair of Boolean Programs with an Application to C”, CAV’06, (Seattle).

[3] Jong-Deok Choi and Andreas Zeller. “Isolating Failure-Inducing Thread Schedules”, (ISSTA 2002), Rome.

[4] Groce, Alex and Chaki, Sagar and Kroening, Daniel and Strichman, Ofer. “Error Explanation with Distance Metrics”, 2005, Springer Verlag, Software Tools for Technology Transfer (STTT).

[5] E. M. Clarke Jr., O. Grumberg, and D. A. Peled. Model checking, MIT Press, 1999.

[6] Bill McCloskey, Feng Zhou, David Gay and Eric Brewer. “Autolocker: Synchronization Inference for Atomic Sections” POPL, Charleston, South Carolina, January 2006.

[7] Chris Lattner and Vikram Adve. ”LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation” Proc. of the 2004 International Symposium on Code Generation and Optimization (CGO’04), Palo Alto, California, Mar. 2004.