Analyzing the Impact of Change in Multi-threaded Programs

Krishnendu Chatterjee[1], Luca de Alfaro[2], Vishwanath Raman[2], and César Sánchez[3]

[1] Institute of Science and Technology, Vienna, Austria
[2] Computer Science Department, University of California, Santa Cruz, USA
[3] IMDEA-Software, Madrid, Spain

Abstract. We introduce a technique for debugging multi-threaded C programs and analyzing the impact of source code changes, and its implementation in the prototype tool DIRECT. Our approach uses a combination of source code instrumentation and runtime management. The source code along with a test harness is instrumented to monitor Operating System (OS) and user defined function calls. DIRECT tracks all concurrency control primitives and, optionally, data from the program. DIRECT maintains an abstract global state that combines information from every thread, including the sequence of function calls and concurrency primitives executed. The runtime manager can insert delays, provoking thread interleavings that may exhibit bugs that are difficult to reach otherwise. The runtime manager collects an approximation of the reachable state space and uses this approximation to assess the impact of change in a new version of the program.

1 Introduction

Multi-threaded, real-time code is notoriously difficult to develop, since the behavior of the program depends in subtle and intricate ways on the interleaving of the threads, and on the precise timing of events. Formal verification provides the ultimate guarantee of correctness for real-time concurrent programs. Verification is however very expensive, and quite often infeasible in practice for large programs, due to the complexity of modeling and analyzing precisely and exhaustively all behaviors. Here, we aim for a more modest goal: we assume that a program works reasonably well under some conditions, and we provide techniques to analyze how the program behavior is affected by software modifications, or by changes in the platform and environment in which the program executes. Our techniques perform *sensitivity analysis* of the code with respect to its environment, and *impact analysis* for software changes [2]. These analyses assist software designers to answer two important questions: (1) *Is the program robust?* Can small changes in platform, compiler options and libraries affect the program's behavior in an important way? (2) *Does a program change introduce unexpected behaviors?*

We propose to instrument a program and run it one or multiple times. At certain points in the program, called *observable statements,* the instrumentation code collects information about the state of the execution, which we call *global state*. Observable statements include OS primitives such as lock and semaphore management, scheduling and timer calls. To perform sensitivity analysis the program is run again, but this time

D.S. Rosenblum and G. Taentzer (Eds.): FASE 2010, LNCS 6013, pp. 293–307, 2010.
© Springer-Verlag Berlin Heidelberg 2010

the instrumentation code simulates changes in the platform. To perform change impact analysis, the versions of source code before and after a change are instrumented and run to compare the set of collected global states. These uses are described in Section 3.

We present the tool DIRECT, which implements these analyses for real-time embedded code written in C, including programs that run on embedded platforms with only limited memory available. The instrumentation stage is implemented relying on the CIL toolset [14]. The effectiveness of DIRECT is shown via two case studies: a C version of dining philosophers, and an implementation of the network protocol 'adhoc' for Lego robots. Specifically, we show how DIRECT can be used to (a) expose a bug in a new version of the adhoc protocol, (b) debug a deadlock in a naive implementation of dining philosophers, (c) compare different fork allocation policies in dining philosophers with respect to resource sharing and equity. We also report on how sensitivity analysis increases thread interleavings and hence the number of unique global states that can be observed for a fixed program and test.

Related work. Change impact analysis is well studied in software engineering [2]. For example, [15,16,17] consider change impact analysis of object-oriented programs written in Java. They use static analysis to determine changes in the implementation of classes and methods and their impact on test suites, to aid users understand and debug failing test cases. Change impact analysis is related to program slicing [19] and incremental data-flow analysis [10]. While there is a large body of work analyzing change impact from the perspective of testing and debugging of imperative, object-oriented and aspect-oriented programs, there is not much literature in change impact analysis for multi-threaded programs. There is work analyzing the impact of change on test selection and minimization [6] and in using runtime information to compute the impact set of a change [9]. Several research efforts study the impact of change based on revision histories. For example, [7] use machine learning techniques to mine source code repositories, trying to predict the propensity of a change in source code to cause a bug.

CHESS [12] explores the problem of coverage in multi-threaded programs attempting to expose bugs by exhaustive thread interleaving. Our work differs from [12] in that we study the impact of change between two versions of a program, whereas CHESS explores only the state space of a single program. Moreover, CHESS borrows techniques from model-checking and it is not easily applicable to the online testing of embedded systems. ConTest [5] explores testing multi-threaded Java programs by placing sleep statements conditionally, producing different interleavings via context switches. ConTest is a Java testing tool, that requires test specification that include the expected outcome. DIRECT does not require test specifications and it is designed to study the impact of change between two versions of a program, while [5] focuses on a single program and test. Moreover, DIRECT targets embedded C programs.

The work closest to ours is [3], that uses runtime, static and source code change information to isolate sections of newly added code that are likely causes of observed bugs. However, [3] does not address concurrent programs, and requires programmer interaction or test specifications to detect "faulty" behavior. In [3] program changes are tracked using information from a version control system. DIRECT accumulates the information at runtime, alleviating the need to rely on sometimes expensive static analysis. This way, we readily obtain a fully automatic tool for embedded systems.

2 Definitions

In this section we present a model of multi-threaded C programs. We consider interleaving semantics of parallel executions, in which the underlying architecture runs a single thread at any given time. This semantics is conventional for most current embedded platforms. The extension to real concurrency (with multi-cores or multi-processors) is not difficult but rather technical, and it is out of the scope of this paper. We now present the formal definitions of our model.

Programs and statements. The dynamics of a program P consist of the execution of a set $T = \{T_i \mid 0 \leq i \leq m\}$ of threads; we take $[T] = \{1, 2, \ldots, m\}$ as the set of indices of the threads in P. Let *Stmts* be the set of statements of P. We distinguish a set of *observable* statements. This set includes all user defined function calls within the user program, as well as all the operating system (OS) calls and returns, where the OS may put a thread to sleep, or may delay in a significant way the execution of a thread. In particular, observable statements include invocations to manage locks and semaphores, such as *mutex_lock*, *semaphore_init* and *thread_delay*. We associate with each statement a unique integer identifier, and we denote by $S \subset \mathbb{N}$ the set of identifiers of all observable statements. We use F to denote the set of all user-defined *functions* in the program and we define $\mathcal{F} : S \mapsto \{\perp\} \cup F$ to be the map that for every statement $s \in S$ gives the function being invoked in s, if any, or $\perp$ if s is not a function call. Finally, we define the *scope* of a statement s to be the user-defined function that contains s, and represent the scope of s as $sc(s)$.

Runtime model. The program is first instrumented with a test harness, and then compiled into a self-contained executable that implements the functionality of the original program together with the testing infrastructure. A *run* is an execution of such a self-contained executable. A *thread state* is a sequence of observable statements $(s_0, s_1, \ldots, s_n)$ where s_n represents an observable statement, and $s_0, s_1, \ldots, s_{n-1}$ the function invocations in the call stack (in the order of invocation) at the time s_n is executed. Precisely, a thread state $\sigma = (s_0, s_1, \ldots, s_n) \in S^*$ is such that each s_0, $\ldots$, s_{n-1} is a call statement and for all $0 < i \leq n$, the scope of s_i is s_{i-1}, that is: $sc(s_i) = \mathcal{F}(s_{i-1})$. In particular, $sc(s_0)$ is the function in which the thread is created, typically $main$. A *block* of code is the sequence of instructions executed between two consecutive thread states. A *joint state* of the program P is a tuple $(k, \sigma_0, \sigma_1, \ldots, \sigma_m, t)$ where,

1. $k \in [T]$ is the thread index of the current active thread,
2. for $0 \leq i \leq m$, the sequence $\sigma_i \in S^*$ is the thread state of the thread T_i, and
3. t is defined as follows: let $\sigma_k = (s_0^k, s_1^k, \ldots, s_n^k)$ be the thread state of the current active thread. If s_n^k is not an OS function call then t is $user$, if s_n^k is an OS function call, then t is $call$ immediately preceding the execution of statement s_n^k, and is ret when the OS function returns.

We refer to the joint states of the program as *abstract global states* or simply as global states. The set of all global states is represented by $\mathcal{E}$.

We illustrate these definitions using Program 1. This program consists of two threads: T_0 that executes $infa$ (on the left); and T_1 that executes $infb$ (on the right). Each thread

Program 1. A simple application with two threads

```
1   void infa(void)  {
2     while (1) {
3       if (exp) {
4         mutex_lock(b);
5         mutex_lock(a);
6         // critical section
7         mutex_unlock(a);
8         mutex_unlock(b);
9      } else {
10        mutex_lock(c);
11        mutex_lock(a);
12        // critical section
13        mutex_unlock(a);
14        mutex_unlock(c);
15      }
16    }
17  }

21  void infb(void) {
22    while (1) {
23      mutex_lock(a);
24      mutex_lock(b);
25      // critical section
26      mutex_unlock(b);
27      mutex_unlock(a);
28    }
29  }
```

is implemented as an infinite loop in which it acquires two mutexes before entering its critical section. The calls $mutex_lock$ and $mutex_unlock$ are the OS primitives that request and release a mutex respectively. For simplicity, assume that the identifier of a statement is its line number. Let s_0 be the statement that launched thread T_0. The state of thread T_0 executing statement $mutex_lock(b)$ at line 4 in function *infa* is $(s_0, 4)$. Similarly, the thread state of T_1 that corresponds to line 23, is $(s_1, 23)$, where s_1 is the statement that launched thread T_1. An example of a global state is $(1, (s_0, 4), (s_1, 23), \mathit{call})$, produced when thread T_1 is in thread state $(s_1, 23)$ and the OS function call at line 23 is about to be executed, indicating mutex a is yet to be acquired, with T_0 being at $(s_0, 4)$. A possible successor is $(1, (s_0, 4), (s_1, 23), \mathit{ret})$, produced when thread T_1 is in thread state $(s_1, 23)$, the OS function call at line 23 has returned, indicating mutex a has been acquired, with T_0 remaining at $(s_0, 4)$.

3 Sensitivity Analysis and Change Impact Analysis

Changes involved during software development and maintenance of concurrent programs can induce subtle errors by adding undesirable executions or disallowing important behaviors. Our goal is to facilitate the discovery of the changes in the behavior of the system due to changes in the source code, or in the execution platform, compiler or libraries. We consider the following sources of differences:

1. *Changes in platform.* When a program is run on a different platform, the execution of each code block may vary due to changes in the target processor.
2. *Changes in compiler options and libraries.* When included libraries or compiler options change, the execution time of each code black may vary.
3. *Source code changes.* Changes in the source code can affect resource interactions and scheduling of the various threads beyond the running time of code blocks.

The goal of DIRECT is to enable the analysis of the above changes, in terms of program behavior. DIRECT operates in two stages. First the system is exercised one or multiple times and the reached states are collected. These runs are called the *reference runs* and the union G of all the reached states is called the *reference set*. Then, a new run R of the program is obtained after the program is affected by some of the above changes. This new run is called the *test run*. The reference set G can be thought of as an approximation of the reachable state space in lieu of a formal specification. If during R a global state e is observed that was not seen in G, DIRECT outputs e along with a trace suffix leading to e. By examining the trace, developers can gain insight into how code changes or environment changes can lead to behavior changes.

Changes in platform, compiler options, and libraries. To analyze the effect of changes in platform, compiler options and libraries, the reference set G and the test run R are obtained from the same program source. G is generated by running the original program with an instrumentation that just collects events. A test run is obtained using DIRECT to modify in an appropriate fashion the duration of the code blocks. This comparison performs *sensitivity analysis,* aimed at discovering the effect of minor timing changes with respect to the reference set. DIRECT can be instructed to modify the block duration in three ways:

- *Proportional delays,* to approximate the effect of changes in platform.
- *Random delays,* to simulate the effect of interrupt handling, included libraries and other characteristics of the hardware.
- *Constant delays,* to simulate the effect of the different latencies of OS calls.

Delay changes can lead to behavior changes in multiple ways. For example, a delay may cause a sleeping thread to become enabled, so that the scheduler can choose this thread to switch contexts. For each of the three delay insertion mechanisms given above, DIRECT can do *selective sensitivity analysis*, where a subset of the threads in the program are subjected to delay insertion.

Changes in source code. To analyze the effects of source code changes, the test run R is obtained using the new version, with or without the injection of delays. For change impact analysis DIRECT compares every global state seen in the test run R of the modified program against the reference set G collected for the original program. Only events in R that correspond to statements of the original program are compared against the set G; events in R corresponding to statements introduced in the new program are trivially not in G. Since we use the set G as an approximation to a formal specification, we are interested in reporting new interleavings with respect to statements that were present in the original program due to source code changes. The instrumentation introduced by DIRECT keeps track of the corresponding statements in the two programs, making a behavioral comparison possible.

Changes in the source code typically involve some change in the logic of the program, brought about by insertion of new code, deletion of code or relocation of some sections of code. In order to analyze the impact of such change between two versions P and P' of a program, it is necessary to relate observable statements corresponding to sections of the code that did not change from P to P'.

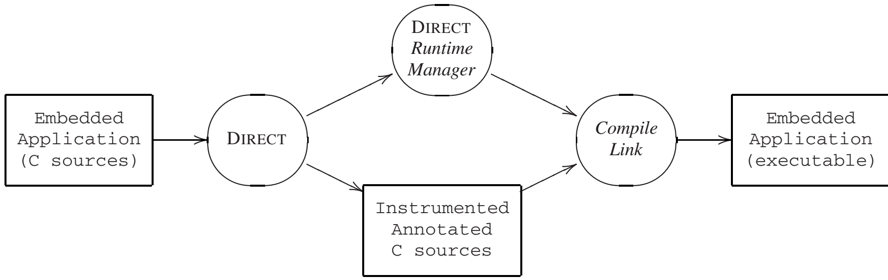


Fig. 1. DIRECT tool flow

Consider again Program 1. If the expression exp in the if condition at line 3 is not always false a deadlock can occur if Thread T_0 acquires resource a and then Thread T_1 acquires resource b. T_1 cannot release resource b until it completes its critical section, which requires resource a held by T_0. One fix for this problem consists in switching the order in which the mutexes a and b are acquired by T_0. Taking line numbers as the identifiers of all observable statements, we notice that the calls to acquire resources a and b are statements 5 and 4 before the change and 4 and 5 after the change. To analyze the impact of this code change, it is necessary to preserve the integer identifiers of these statements during program transformation, even though these statements have moved in the course of the transformation.

4 Implementation

We discuss now the relevant implementation issues.

4.1 Program Instrumentation

Fig. 1 shows the program instrumentation flow of DIRECT. DIRECT relies on the CIL toolset [14] to parse and analyze the program under consideration, and to insert instrumentation in the code. The instrumented version of the program is compiled and linked with a runtime manager to produce the final executable. The application can then be run just like the original user program. The runtime manager is a custom piece of software that gains control of the user application before and after each observable global state. The instrumentation step performs two tasks:

- replace observable statements with appropriate calls to the resource manager, allowing the tracking of visible statements and the insertion of delays.
- wrap every call to a user defined function with invocations to the run-time manager that keep track of the call stack.

Instrumenting observable statements. DIRECT reads a configuration file that specifies the set of functions to track at runtime. This set typically includes OS primitives such as mutex and semaphore acquisitions and releases, and other timing and scheduling-related primitives. Each observable statement s is replaced by a call to a corresponding

Program 2. man_mutex_lock replaces mutex_lock in the source code

```
void man_mutex_lock  (int statement_id, resource_t a)  {
  // Gets the current thread id from the set of registered threads.
  int thread_id = self_thread_id();

  // Injects pre-call delays for sensitivity analysis.
  injectDelay(thread_id, Pre);

  // Generates \ProgramEvent before the OS function call.
  registerJointState(thread_id, statement_id, call);

  // Calls the actual OS primitive.
  mutex_lock(a);

  // Injects post-call delays for sensitivity analysis.
  injectDelay(thread_id, Post);

  // Generates \ProgramEvent after the OS function call.
  registerJointState(thread_id, statement_id, ret);

  // Stores the start time of the subsequent block of code.
  storeBlockStartTime(thread_id);
}
```

function in the runtime manager. The function in the runtime manager performs the following tasks:

1. First, an optional delay can be introduced to simulate a longer run-time for the code block immediately preceding the observable statement.
2. The internal representation of the thread state is updated, due to the occurrence of the observable statement s.
3. The original observable statement s (such as an OS call) is executed.
4. An optional delay can be introduced, to simulate a longer response time from the OS, or the use of modified I/O or external libraries.
5. Finally, the internal representation of the thread state is again updated, indicating the completion of the statement s.

Note that DIRECT updates the thread state twice: once before executing s, another when s terminates. Distinguishing these two states is important. For example, when the thread tries to acquire a lock, the call to *mutex_lock* indicates the completion of the previous code block and the lock request, while the completion of *mutex_lock* indicates that the thread has acquired the lock. Program 2 illustrates the implementation of the runtime manager function *man_mutex_lock* that replaces the OS primitive *mutex_lock*. The first argument in all calls to runtime manager functions that replace OS functions is $s \in S$. The subsequent arguments are the actual arguments to be passed to the OS primitive.

Tracking thread states. DIRECT also tracks the call stack to perform context-sensitive analysis, distinguishing calls to the same function that are performed in different stack

configurations. To this end, DIRECT wraps each function call in a *push*-*pop* pair. If i is the integer identifier of the call statement, the *push* instrumentation call adds i to the call stack, and the *pop* call removes it.

Preserving accurate timing. The instrumentation code, by its very existence, causes perturbations in the original timing behavior of the program. To eliminate this undesirable effect, DIRECT freezes the real-time clock to prevent the runtime processing overhead from affecting the timing of the application code. The current version of DIRECT implements this freezing as a modified version the Hardware Abstraction Layer (HAL) in the eCos synthetic target running on Ubuntu 8.04. In this manner, the exposed bugs are not caused by artificial interleavings created by the effect of the runtime manager, and they are more likely to correspond to real bugs.

Tracking corresponding pieces of code. To perform change impact analysis, it is important to identify the common, unchanged portions of P and P'. A transformation from P to P' may involve (a) sections of new code that are inserted, (b) sections of code that are deleted, and (c) sections of code that have moved either as a consequence of insertions and deletions or as a consequence of code re-organization. DIRECT deals with these variations by first generating a text dump summarizing the CFG of P and P'. The key problem in tracking code changes is that of variations in coding style; syntactically identical program fragments may still be very different based on the use of indentation, line breaks, space characters and delimiters. Our CFG summaries preserve instructions (assignments and function calls) exactly, but summarize all other statements (blocks, conditionals, goto statements etc.) This summarization is done to remove artifacts such as labels introduced by CIL that may change from P to P', but have no bearing on tracking statements. Fig. 2 shows the summary generated for the program fragment on the left of Program 1. Given two CFG summaries, DIRECT identifies sections of code that have been preserved using a text difference algorithm [18,13,4]. Given two text documents D and D', this algorithm extracts a list of space, tab and newline delimited words from each document. The list of words are compared to produce a set of insertions, deletions and moves that transform D to D'. We use the set of moves generated by the algorithm to relate the set of statements in P that are also in P'.

```
<Block>
<Loop>
<If>
<Block>
  cyg_mutex_lock(& a);
  cyg_mutex_lock(& b);
  cyg_mutex_unlock(& b);
  cyg_mutex_unlock(& a);
<Block>
  cyg_mutex_lock(& c);
  cyg_mutex_lock(& a);
  cyg_mutex_unlock(& a);
  cyg_mutex_unlock(& c);
```

Fig. 2. A summary snippet

Tracking additional components of the program joint state. DIRECT supports the following extensions to the joint state of a program.

- *Resource values.* Resources are often managed and synchronized using concurrency control primitives. Since DIRECT captures these control primitives the precise values of the resources can be accessed by the runtime manager with total

precision. Let R be the set of all *resources*, including mutexes and semaphores. Every resource has an associated *value*, that has the range $\{0, 1, 2, \ldots, \max(r)\}$, where $\max(r) = 1$ for all mutexes and $\max(r) > 0$ for all counting semaphores.
- *Global variables.* DIRECT can also track global variables, but these values are not tracked whenever they change but only when an observable statement is reached.
- *Extending observable statements.* Users can expand on the set of OS primitives or library functions to track.
- *Block execution times.* Average block execution times of each block in each thread can be tracked to later perform proportional delay injection.

4.2 Detecting New Events Efficiently

To perform sensitivity and change impact analysis, it is crucial to test efficiently whether an observed event is a member of a given state set. Time efficiency is needed to scale to large programs. Space efficiency is especially important in the study of embedded software. Even though the set of states represented can be very large, an embedded software implementation can only use a very limited amount of memory. To achieve the desired efficiency, DIRECT stores the set of reachable states as a Bloom filter [11], a probabilistically correct data-type that implements sets of objects, offering two operations: insertion and membership checking.

Bloom filters guarantee that after inserting an element, checking membership of that element will return true. However, a membership query for an element that has not been inserted in the Bloom filter is only guaranteed to respond false (the correct answer) with a high probability. That is, Bloom filters allow some false positive answers for membership queries. This fact implies that DIRECT may (rarely) miss new global states, but that every new global state found by DIRECT is guaranteed to be new.

The performance of Bloom filters depends on the use of good (independent) hash functions, which are difficult to design. DIRECT uses double-hashing [8] to obtain k (good) hash functions from 2 (good) hash functions. Therefore, the cost of an operation is virtually that of the computation of two hash-functions, so all operations run in almost constant time.

5 Case Studies

We report two case studies: a solution to the dining philosophers problem and an adhoc protocol for legOS, adapted to run in an eCos [1] environment.

5.1 An Adhoc Protocol

We analyzed a multi-threaded implementation of an ad-hoc network protocol for Lego robots. As illustrated in Fig. 3, the program is composed of five threads, represented by ovals in the figure, that manage four message queues, represented by boxes. Threads *user* and *generator* add packets to the *input* queue. Thread *router* removes packets from the *input* queue, and dispatches them to the other queues. Packets in the *user* queue are intended for the local hardware device and hence are consumed by the *user* thread. Packets in the *broadcast* queue are intended for broadcast, and they are moved

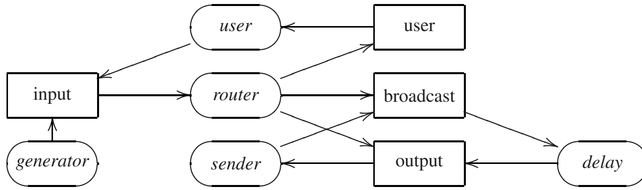


Fig. 3. Scheme of an ad-hoc network protocol implementation

Program 3. A snippet of code from the packet router thread

```
1    semaphore_wait(&bb_free_sem);
2    semaphore_wait(&bb_mutex);
4    // code that forms a new packet and copies it into
5    // the free slot in the broadcast queue
     ...
40   semaphore_post(&bb_mutex);
     ...
50   semaphore_post(&bb_els_sem);
     ...
```

to the *output* queue by the *delay* thread, after a random delay, intended to avoid packet collisions during broadcast propagation. Packets in the *output* queue are in transit to another node, so they are treated by the *sender* thread. Notice that if the *sender* fails to send a packet on the network, it reinserts the packet back in the *broadcast* queue (even if it is not a broadcast packet), so that retransmission will be attempted after a delay. Each queue is protected by a mutex, and two semaphores that count the number of empty and free slots, respectively. The reference implementation has no non-blocking resource requests. Program 3 shows a snippet of the router code. It first checks whether the broadcast queue is free by trying to acquire the semaphore *bb_free_sem* at line 1. If the semaphore is available, the router acquires a mutex, *bb_mutex* that controls access to the broadcast queue, before inserting a packet in the queue. Then, the router posts the semaphore *bb_els_sem* indicating that the number of elements in the queue has increased by one.

A very subtle bug is introduced by replacing the call to acquire the semaphore *bb_free_sem* by a non-blocking call. The change is itself quite tempting for a developer as this change improves CPU utilization by allowing the router not to block on a semaphore, continuing instead to process the input queue while postponing to broadcast the packet. Program 4 shows the snippet of code that incorporates this change. The bug is exhibited when the the block of code that should execute when the semaphore is successfully acquired, terminates prematurely. Specifically, the call to post the semaphore *bb_els_sem* at line 51 should only occur when the call at line 1 to acquire *bb_free_sem* succeeds. This bug goes undetected as long as the call to acquire *bb_free_sem* always succeeds. Two other threads, besides the router, access the semaphore *bb_free_sem*: the delay thread and the send thread. Notice that as long as the send thread succeeds, it does not try to place the packet back on the broadcast queue and the bug goes undetected. If

Program 4. The router thread after changing to a non-blocking call (trywait)

```
1    if (semaphore_trywait(&bb_free_sem)) {
2      semaphore_wait(&bb_mutex);
4      // code that forms a new packet and copies it into
5      // the free slot in the broadcast queue
       ...
40     semaphore_post(&bb_mutex);
41   }
     ...
51   semaphore_post(&bb_els_sem);
     ...
```

the send thread fails to send the packet, acquires *bb_free_sem* and causes the broadcast queue to fill up, the router fails to get *bb_free_sem*, exposing the bug that eventually leads to a deadlock, where packets can no longer be routed. In one of our tests for this program, we model failure to send a packet using randomization; each attempt to send a packet has an equal chance at success and failure. This test exposed the bug. Specifically, the new global state observed corresponds to an invocation to post *bb_els_sem* at line 51 in Program 4. In the Appendix, we show the last two global states in the suffix of states that lead to this new state. The global state immediately preceding the new state is one where the non-blocking semaphore request in the packet router fails. DIRECT reports a trace to the new global state (the bug) and tools to visualize this trace.

5.2 Dining Philosophers

Program 5 shows an implementation of a philosopher in a proposed solution to the dining philosophers problem analyzed using DIRECT. The numbers on the left are identifiers of observable statements. A naive implementation lets each philosopher pick up her left fork first leading to a deadlock; each philosopher is holding her left fork and none can get an additional fork to eat. Table 1, shows the tail-end of the sequence of states of a system with 5 dining philosophers. Each line shows a global state containing the index of the active thread, the state of each thread, the resource values, the set of resources held by the active thread and whether the event corresponds to a function call or return. The transition from state 5 to state 6 is the one where the fifth philosopher (thread T_4) acquires her left fork. As evidenced in state 6 all resources have been allocated with each philosopher holding one fork. This state inevitably leads to a deadlock, shown in the final state, where all philosophers are waiting at statement 3, that corresponds to a request for the second fork in Program 5. When we fix the deadlock using monotonic locking and run the program again, we notice that the new state is one where the fifth philosopher is denied her first fork, avoiding the deadlock.

We found that the sequences of states generated serve another useful purpose, namely analyzing waiting times for philosophers and checking whether the fork allocation policies are philosopher agnostic. We analyzed the sequence of states generated after fixing the deadlock. We noticed that a simple analysis on the sequence shows that the observable statement 4 where the philosophers have acquired both forks, occurs half the number of times for the first and last philosophers compared to the others. If we change

Program 5. Dining philosopher

```
  void philosopher(int philosopher_id) {
    int first_fork, second_fork;

    // fork assignment policy
    first_fork = philosopher_id;
    second_fork = (philosopher_id + 1) % N_PHILS;

    if (first_fork > second_fork) {
      first_fork = second_fork;
      second_fork = philosopher_id;
    }
    while (1) {
 0    thread_delay(20);    // thinking phase

      // eating phase
 1    semaphore_wait(&forks[first_fork]);  // pick first fork
 2    thread_delay(2);                     // pause
 3    semaphore_wait(&forks[second_fork]); // pick second fork
 4    thread_delay(20);                    // eating phase
 5    semaphore_post(&forks[second_fork]); // replace second fork
 6    thread_delay(2);                     // pause
 7    semaphore_post(&forks[first_fork]);  // replace first fork
    }
  }
```

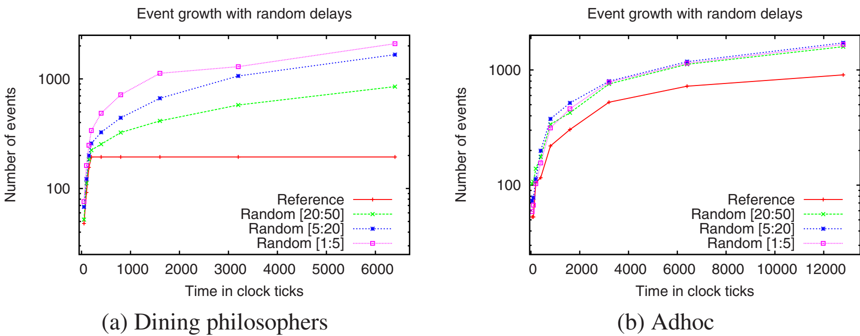
the implementation so that all the even numbered philosophers pick their left fork first and the odd numbered philosophers pick their right fork first, they all get to eat virtually the same number of times. The latter implementation may cause livelocks under certain schedulers, but is equitable to the philosophers when compared to monotonic locking. The asymmetry in the implementation for the last philosopher turns out to be the culprit. Since the last philosopher always wishes to pick up her right fork first which is also the first fork that the first philosopher needs, they end up waiting for each other to finish. The last philosopher cannot pick up her left fork till she gets her right fork and vice versa for the first philosopher. This asymmetry favors the other philosophers. In fact, philosophers T_0 and T_4 acquire their first fork roughly half the number of times that the others do, and have the largest wait times for their forks when compared to the others.

5.3 Increasing Coverage with Random Delays

An interesting question in change impact analysis is that of coverage. Given a program, a test and a platform, how do we generate as many global states as possible? The larger the number of states that the tool exercises, the more likely it is that a state observed in a test run, that does not occur in any reference run, point to a potential bug or otherwise interesting new global state. Towards this end, DIRECT provides the mechanism of injecting random delays, in user specified ranges, that increases context switching

Table 1. A sequence leading to a deadlock in a naive implementation of dining philosophers

State no.	*Thread index*	Philosopher threads T_0	T_1	T_2	T_3	T_4	*Res values*	*Res held*	*(c)alls/ (r)ets*
1	3	$(0,2)$	$(0,2)$	$(0,2)$	$(0,1)$	$(0,0)$	$(0,0,0,1,1)$	()	c
2	3	$(0,2)$	$(0,2)$	$(0,2)$	$(0,1)$	$(0,0)$	$(0,0,0,0,1)$	(3)	r
3	3	$(0,2)$	$(0,2)$	$(0,2)$	$(0,2)$	$(0,0)$	$(0,0,0,0,1)$	(3)	c
4	4	$(0,2)$	$(0,2)$	$(0,2)$	$(0,2)$	$(0,0)$	$(0,0,0,0,1)$	()	r
5	4	$(0,2)$	$(0,2)$	$(0,2)$	$(0,2)$	$(0,1)$	$(0,0,0,0,1)$	()	c
6	4	$(0,2)$	$(0,2)$	$(0,2)$	$(0,2)$	$(0,1)$	$(0,0,0,0,0)$	(4)	r
...	...								
16	4	$(0,3)$	$(0,3)$	$(0,3)$	$(0,3)$	$(0,2)$	$(0,0,0,0,0)$	(4)	r
17	4	$(0,3)$	$(0,3)$	$(0,3)$	$(0,3)$	$(0,3)$	$(0,0,0,0,0)$	(4)	c



(a) Dining philosophers (b) Adhoc

Fig. 4. Number states observed as running time increases, with and without random delays

between threads, producing new states. We studied the effect of injecting random delays in the ranges [1..5], [5..20] and [20..50] clock ticks in all threads for the two case studies presented in this section. We plot the results in Figure 4a and Figure 4b, where the x-axis represents run durations in clock ticks and the y-axis reports the log of the number of unique states observed. In this study, we ran each program for a set of durations. In each run, we first measured the number of unique states without random delay injection; the *Reference* line in the graphs. For each duration, we then ran the same application, injecting random delays and took the union of the set of states seen in the reference run and the set of states seen with random delay injection. The size of these sets, for each run duration, are shown by the points along the lines labeled with random delays in the graphs. For dining philosophers, we noticed that the number of *new* states in the reference run is zero after 200 clock ticks, but using random delays we see an increase in the number of new states for each run duration as shown in Figure 4a. Since code blocks take longer to execute with delay injection, the total number of global states diminishes with longer delays, reducing the number of unique states seen. This phenomenon is also witnessed by the increase in states seen with smaller delay ranges. For the adhoc protocol, we noticed that the number of new states observed in the reference run decreases as the duration of the runs increase, but the number of new states are

consistently higher with random delay injection just as in the case of dining philosophers. We also observed that in this case, changing the range of the random delays does not produce any significant change in the number of new states seen, unlike in the case of dining philosophers. These results on our case studies give us strong evidence that random delay injection is a good mechanism to increase the number of observed states for a given program and test. We note here that the techniques proposed in CHESS [12] or [5] can be used in addition to random delay injection to get a better approximation of the reachable state space.

6 Conclusions

This paper reports on techniques for the change impact and sensitivity analysis of concurrent embedded software written in the C programming language. These techniques are prototyped in a tool called DIRECT, that uses a combination of static analysis and runtime monitoring. The static analysis determines instrumentation points, generates the monitoring code, and establishes the difference between two versions of a given program. The runtime manager is executed before and after every concurrency primitive and user defined function, and computes at each instrumentation point an abstraction of the current state. The runtime manager also keeps a sequence of abstract global states leading to the current state.

For sensitivity analysis, the runtime manager also inserts delays to simulate differences between platforms, libraries and operating systems. For change impact analysis, the runtime manager collects an approximation of the set of reached states of the original program. Exhaustive exploration techniques like [12,5] can approximate more accurately the reachable state space, but they are not directly suitable for embedded systems, and to perform (online) change impact analysis. The states reached during the executions of the new version are then compared against the set of reached states of the original program. The prototypes were developed in a modified version of the eCos environment in which the instrumented code was executed with the real-time clock stopped, so that the execution of the runtime manager incurred no additional delay. We presented two case studies to illustrate how the techniques described in this paper can help capture bugs in concurrency programs.

There are some limitations of the work presented here that we plan to study in future research. First, we would like to apply DIRECT to large programs to see how well our techniques scale with program size. Second, the indication of a new global state may not correspond to a real bug but just a false positive. While we did not encounter such false positives in our case studies, we plan to study the number of false positives as the program size increases and steps to minimize them. Finally, the current sensitivity analysis can only insert delays in execution blocks. We would like to extend it with the ability to accelerate blocks (negative delays), whenever it is safe to do so.

We also plan to extend the techniques reported here in two directions. First, we will use DIRECT in real embedded systems, where the illusion of instantaneous execution time of the manager that we obtained via simulation is not accurate. Second, we will explore the design of schedulers that try to maximize the set of global states reached. Unlike in CHESS [12] we plan to proceed in several rounds, where the scheduler of the

next round is obtained using the set of global states obtained in the previous runs with some static analysis.

References

1. ecos homepage, `http://ecos.sourceware.org/`
2. Arnold, R.S.: Software Change Impact Analysis. IEEE Computer Society Press, Los Alamitos (1996)
3. Bohnet, J., Voigt, S., Döllner, J.: Projecting code changes onto execution traces to support localization of recently introduced bugs. In: Proc. of the 2009 ACM Symposium on Applied Computing (SAC 2009), pp. 438–442. ACM, New York (2009)
4. Burns, R.C., Long, D.D.: A linear time, constant space differencing algorithm. In: Performance, Computing, and Communication Conference (IPCCC 1997), pp. 429–436. IEEE International, Los Alamitos (1997)
5. Edelstein, O., Farchi, E., Goldin, E., Nir, Y., Ratsaby, G., Ur, S.: Framework for testing multithreaded Java programs. Concurrency and Computation: Practice and Experience 15(3-5), 485–499 (2003)
6. Harrold, M.J.: Testing evolving software. Journal of Systems and Software 47(2-3), 173–181 (1999)
7. Kim, S., James Whitehead, J.E., Zhang, Y.: Classifying software changes: Clean or buggy? IEEE Transactions on Software Engineering 34(2), 181–196 (2008)
8. Knuth, D.E.: The Art of Computer Programming, 2nd edn. Sorting and Searching, vol. 3. ch. 6.4. Addison-Wesley, Reading (1998)
9. Law, J., Rothermel, G.: Whole program path-based dynamic impact analysis. In: ICSE 2003, pp. 308–318 (2003)
10. Marlowe, T.J., Ryder, B.G.: An efficient hybrid algorithm for incremental data flow analysis. In: Proc.of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1990), pp. 184–196. ACM, New York (1990)
11. Mitzenmacher, M., Upfal, E.: Probability and Computing. Cambridge University Press, Cambridge (2005)
12. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing Heisenbugs in concurrent programs. In: OSDI 2008, pp. 267–280 (2008)
13. Myers, E.W.: An O(ND) difference algorithm and its variations. Algorithmica 1(2), 251–266 (1986)
14. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Infrastructure for C program analysis and transformation. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002)
15. Ren, X., Chesley, O.C., Ryder, B.G.: Identifying failure causes in Java programs: An application of change impact analysis. IEEE Trans. Softw. Eng. 32(9), 718–732 (2006)
16. Ren, X., Shah, F., Tip, F., Ryder, B.G., Chesley, O.: Chianti: a tool for change impact analysis of Java programs. SIGPLAN Not. 39(10), 432–448 (2004)
17. Ryder, B.G., Tip, F.: Change impact analysis for object-oriented programs. In: PASTE 2001: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, pp. 46–53. ACM, New York (2001)
18. Tichy, W.F.: The string-to-string correction problem with block move. ACM Trans. on Computer Systems 2(4) (1984)
19. Tip, F.: A survey of program slicing techniques. J. Prog. Lang. 3(3) (1995)