Analyzing the Impact of Change in Multi-threaded Programs*

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

[1] School of Engineering, University of California, Santa Cruz, USA
[2] IMDEA-Software, Madrid, Spain

Technical Report UCSC-SOE-09-25
School of Engineering, University of California, Santa Cruz
August 2009

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. All concurrency control primitives are tracked through the OS functions. Optionally, Direct can track some concurrency related data of interest. Direct keeps track of an abstract global state that combines the abstract states of 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, and that may be 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 the events that affect the code. 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]. In particular, we aim at discovering changes in program behavior due to:

* This work has been partially supported by NSF CCR-0132780, NSF CCR-0234690

- *Changes in platform,* which can affect the execution time of a code block, according to the instruction set, cache, and CPU speed of the target platform.
- *Changes in compiler options and libraries,* which can affect execution speed as well.
- *Source code changes,* which can affect not only the execution time of the code blocks, but also the resource interaction and scheduling of the various threads.

Our analysis assist designers to answer two important questions:

- *Is the program robust?* Can small changes in the scheduler, in CPU speed, in platform, or in libraries cause the program to break? How much does the behavior of the program depend on the particular timing of the platform on which it is tested?
- *Does a program change introduce unexpected behaviors?* Every program change is likely to produce new behaviors, but does a change lead to new interleavings, and possibly new bugs, in the system? To what extent does a code change affect the concurrency behavior of a program?

In order to perform this sensitivity and change impact analysis, we propose to instrument a program P and run it one or multiple times. The instrumentation tracks the behavior of P in terms of *global states* that summarize the joint state of all threads with respect to a set of *observable statements.* These observable statements include OS primitives such as lock and semaphore management, scheduling calls, timer calls, and more. The outcome of a first round of runs is a set E of global states which approximates the reachable state space. Once this round finishes, we run an instrumented program P', obtained from P in one of two ways: by adding delays that simulate changes in platform, compiler options, or libraries, or by including in P' software changes that have been proposed for P. Whenever a run of P' encounters a global state that has not been seen previously for P (i.e., not in E), we output the new global state, along with a trace that leads to it. This output information alerts designers to the possible arousal of new behaviors in P', and offers debugging information that can be used to decide whether the new behaviors are harmful.

We present the tool Direct, which implements these analyses for real-time embedded code, including programs that run on embedded platforms with only limited memory available. Direct can be applied to programs written in C. The instrumentation stage is implemented relying on the CIL toolset [12]. The instrumentation code keeps track of the global states encountered during execution, and can be used to modify the program timing. For sensitivity analysis, the program is run twice, using the instrumentation to simulate the effects of platform, libraries, and scheduling changes. For change impact analysis, Direct first identifies the changes in the two programs, and tracks the common portions of code across the programs. Direct then instruments both original and modified programs in a compatible fashion, so the outcomes of the two instrumented programs can be compared.

To make such an analysis feasible, it is essential (a) to represent the set E of global states encountered in the first set of runs in a space-efficient fashion, and (b) to efficiently test whether a newly encountered global state belongs to E or not. The requirement (a) is particularly important in the analysis of embedded software, where the set E must be kept in what often is a very small amount of working memory. DIRECT represents the set E using Bloom filters [9], a probabilistic data-structure that implements sets efficiently. The fact that Bloom filter membership has false positives, but no false negatives, implies that DIRECT may (rarely) miss new global states, but that every new global state found by DIRECT is guaranteed to be new.

We demonstrate the effectiveness of DIRECT via case studies. We studied implementations of the dining philosophers problem and a program that implements an adhoc protocol for Lego robots. We studied capabilities of the DIRECT infrastructure to, (a) expose a plausible bug introduced as a consequence of a likely change in the implementation of the adhoc protocol, (b) study the usefulness of the sequence of global states extracted by DIRECT to 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, (d) study the usefulness of sensitivity analysis to increase thread interleavings and hence increase 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, [13–15] consider change impact analysis of object-oriented programs, specifically Java programs. They use static analysis to determine additions, deletions and modifications to classes and methods and their impact on test suites, with the objective of aiding users understand and debug failing testcases. Change impact analysis is also related to program slicing [17] and incremental data-flow analysis [8]. While there are a number of works in analyzing change impact from the perspective of testing, debugging and test case generation of imperative, object-oriented and aspect-oriented programs, there is not much literature devoted to change impact analysis in multi-threaded programs.

There are several works devoted to predicting the impact of change based on revision histories. For example, [6] studies the analysis of change impact using machine learning techniques, mining source code repositories, to predict the propensity, of a change in source code, to cause a bug. CHESS [10] explores the problem of coverage in multi-threaded programs to expose bugs caused by unexplored thread interleavings. Our work differs from [10] in that we study the impact of change between two version of a program, whereas CHESS explores only the state space of a single program. The work in [5] explores testing multi-threaded Java programs with the objective of finding bugs, by placing sleep statements conditionally, thus producing different interleavings via context switches. The tool reported in [5] is a Java testing tool, that requires every test to have a specification of its correct outcome; the tool repeats each test a number of times modifying the concurrent behavior. Our work differs from [5] in that we do not require test specifications and we focus on the problem of

assessing the impact of change between two versions of a program, while [5] focusses on a single program and test. Moreover, we target 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 require programmer interaction or test specifications to detect "faulty" behavior. In [3] program changes are tracked, using information from a version control system to generate the set of changes that affected the faulty behavior. On the other hand, our work starts by accumulating sets of global states at runtime for a given set of tests. Then, we explore whether events not seen in earlier executions are caused by changes in the program, as witnessed by running the same set of tests. By restricting ourselves to runtime behaviors, we alleviate the need to rely on sometimes expensive static analysis, and we readily obtain a fully automatic tool.

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 during the execution of the programs. 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. Our goal is to provide a debugging tool for multi-threaded C programs by using a combination of source instrumentation and online run-time analysis. We now present the formal definitions of our model.

Programs and statements. The dynamics of a multi-threaded C program P consists of the execution of a set $T = \{T_i \mid 0 \leq i \leq n\}$ of threads; we take $[T] = \{1, 2, \ldots, n\}$ as the set of indices of the threads in P. Given the program P, let *Stmts* be the set of statements of P. We distinguish a set of *observable* statements. This set includes all 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, the observable statements include the invocations to manage locks and semaphores and change thread interleavings, 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* being called in the program and we define $\mathcal{F} : S \mapsto \{\bot\} \cup F$ to be the map that for every statement $s \in S$ gives the function being invoked in s, if any, or $\bot$ if s is not a function call. Finally, we define the *scope* of a statement $s \in S$ to be the function that contains s. We 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 contains 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* $(s_0, s_1, \ldots, s_n)$ consists

Program 1 A simple application with two threads

```
1   void infa(void)
2   {
3     while (1) {
4       if (exp) {
5         mutex_lock(b);
6         mutex_lock(a);
7         // critical section
8         mutex_unlock(a);
9         mutex_unlock(b);
10      } else {
11        mutex_lock(c);
12        mutex_lock(a);
13        // critical section
14        mutex_unlock(a);
15        mutex_unlock(c);
16      }
17    }
18  }
```

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

of an observable statement s_n, together with all the statements $s_0, s_1, \ldots, s_{n-1}$ that correspond to the function calls 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, $sc(s_0) = main$, and for all $0 < i \leq m$, the scope of s_i is s_{i-1}: $sc(s_i) = \mathcal{F}(s_{i-1})$. 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 $(k, \sigma_0, \sigma_1, \ldots, \sigma_n)$, where $k \in [T]$ is the thread index of the current active thread, and for $0 \leq i \leq n$, the sequence $\sigma_i \in S^*$ is the thread state of the thread T_i. 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 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 in the explanation, assume that the identifier of a statement is its line number. Let s_0 be the statement that launched thread T_0. The thread state of T_0, corresponding to the statement at line 5 in function *infa* that calls $mutex_lock(b)$ is $(s_0, 5)$. Similarly, the thread state of T_1 corresponding 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 $(1, (s_0, 5), (s_1, 23))$, produced when thread T_1 transitions to state $(s_1, 23)$ with T_0 being at $(s_0, 5)$, corresponding to its last visited statement.

3 Sensitivity Analysis and Change Impact Analysis

The software development and maintenance processes of concurrent programs involve changes that can induce subtle errors by adding undesirable executions

or disallowing important behaviors. The goal of our work is to facilitate the discovery of the changes introduced in the behavior of the system due to changes in the source code, or changes in the platforms, compiler, and libraries that allow the construction of a working system from the source code. Thus, our work spans both *sensitivity analysis* and *change impact analysis.* 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 instruction set, cache, and CPU speed of the target machine.
2. *Changes in compiler options and libraries.* When the included libraries or compiler options change, the execution time of each code black may vary, with some executing faster and others slower.
3. *Source code changes.* When the source code of a program changes, it can affect not only the execution time of the code blocks, but may also change resource interactions and scheduling of the various threads.

The goal of DIRECT is to enable developers to analyze the effect of the above changes, in terms of program behavior.

DIRECT first computes an approximation G of the set of global states that the program can reach by collecting the states that the program reaches in one or multiple runs. Then, a new run R of the program is obtained after the program is affected by some of the above changes. When the run R encounters a global state e that has never been observed in G, DIRECT outputs e along with a trace suffix leading to e. The developers can then examine the trace, and gain insight into how code 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 set G and the run R used in the comparison are obtained from the same program source. The run R is generated by running the original program without instrumentation; the set G is obtained using DIRECT to modify in an appropriate fashion the duration of the code blocks. This comparison can be used to perform *sensitivity analysis,* aimed at discovering how much the behavior of the original program can be affected by minor timing changes. DIRECT can be instructed to modify the duration of code blocks in three ways:

- *Proportional delays.* The effect of changes in platform can be approximated by introducing delays, for each block of code, that are proportional to the running time of the block itself.
- *Random delays.* Even on the same execution platform the running time of a block can vary due to characteristics of the hardware, the handling of interrupts, included libraries, etc. These effects can be simulated by injecting random delays in code blocks, within a user specified range of values. This technique also aids a programmer in studying how robust an implementation is against changes in block execution times.
- *Constant delays.* The latency of different instances of OS calls can also vary. This effect can be simulated by inserting constant time delays in the testing environment.

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 program event before the OS function call.
  registerJointState(thread_id, statement_id);

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

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

  // Generates program event after the OS function call.
  registerJointState(thread_id, statement_id + 1);

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

Delay changes can lead to behavior changes in multiple ways. For example, a block that is delayed may result in a thread to become enabled, so that the scheduler can choose to switch contexts from the delayed thread to the one that is enabled; a shorter delay may not present such an option for the scheduler. 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, DIRECT uses sensitivity analysis through the injection of delays to accumulate such a set of global states for the original program. As we show in subsection 5.3, this is an effective mechanism to collect a large set of global states for a given program and test. We then compare global states seen in a run R of the modified program against the set G of global states collected for the original program. 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, such as 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'.

To motivate the need to track observable statements before and after a change in code, consider again the example in Program 1. If the expression *exp* in the if condition is not always false in Thread 1, there exists a potential for a deadlock. A

deadlock can occur if Thread 1 acquires resource a and then Thread 2 acquires resource b. Thread 2 cannot release resource b until it completes its critical section, which requires resource a held by Thread 1. At this point, there is a deadlock. One fix for this problem consists in switching the order in which the mutexes a and b are acquired by Thread 1. 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 6 before the change and 6 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

In order to perform sensitivity analysis or change impact analysis for a program P using Direct, we proceed as follows. First, we use the Direct tool to produce an instrumented version of the original program. This version is then run one or multiple times, producing a set $E \subseteq \mathcal{E}$ of observed global states.

For sensitivity analysis, we then run P instructing Direct to insert delays as detailed in Section 3. For change impact analysis, we run a modified (and instrumented) version P' of the program, possibly modifying timing as well. In either case, the goal is to uncover states visited in these runs that do not belong to E. When such an state is reached, Direct outputs the sequence of the m most recent global states leading to the new state; the parameter m is chosen to balance the need to limit memory consumption, and the intent to provide insightful debugging information.

4.1 Program Instrumentation

Fig. 1 shows the program instrumentation flow of Direct. Direct relies on the CIL toolset [12] to parse and analyze the program under consideration; CIL is also used 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. In the following subsections, we describe the instrumentation phase, the runtime manager and how we track change in source code across multiple revisions. The instrumentation step performs two tasks:

- replace observable statements, such as OS calls, 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 to keep track of the call stack of the program being run.

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/releases, and other

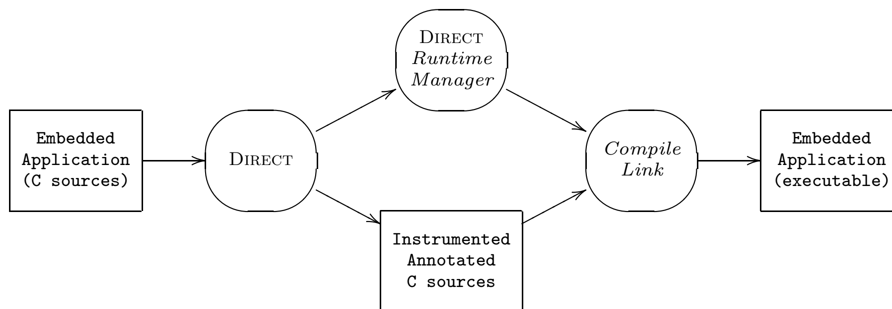


Fig. 1: DIRECT tool flow. From the embedded program sources to the final executable after instrumentation and linking with the runtime manager.

timing and scheduling-related primitives. Each observable statement s is replaced by a call to a corresponding function in the runtime manager. The function in the runtime manager performs the following tasks:

1. First, an optional delay can be introduced. This delay simulates a longer runtime 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. This delay simulates 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 we update 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. In Program 2, we illustrate 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 passed to the OS primitive in the user program; these are used to call the actual primitive inside the runtime manager.

Tracking thread states. In order to track thread states, in addition to the observable statements, DIRECT also tracks the call stack of the program. This information is used 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. This bookkeeping allows the runtime manager to produce the thread states correctly when observable statements are encountered.

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, we run all applications on the eCos synthetic target running on Ubuntu 8.04. We modified the Hardware Abstraction Layer (HAL) by adding infrastructure to freeze the real-time clock, so that all of the runtime processing overheads do not affect the timing of the application code. 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 that can be observed in the target environment. Notice that while the freezing of the real-time clock is a functionality that can be implemented only when we run the user program on an emulator, such as the eCos HAL, the observed bugs remain real bugs in the actual environment, modulo accuracy of emulation.

Tracking corresponding pieces of code. In change impact analysis, DIRECT is used to analyze the difference in behavior between a program P, and a program P' obtained from P by modifying some of its source code. To perform this analysis, it is important to identify the common, unchanged portions of P and P'. In particular, we must preserve the identifiers of statements in P that are also present in 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. 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. A mechanism that compares a program P and its transformed version P' needs to compare a representation of P and P' that remove these variations in coding style. A representation that lends itself to such comparison is the CFG. Hence, in DIRECT, we first generate a text dump summarizing the CFG of P and P'. In our CFG summaries we preserve instructions (assignments and function calls) exactly, but summarize all other statements (blocks, conditionals, goto statements etc.). This is done for two reasons; firstly, to remove artifacts such as labels introduced by CIL that may change from P to P', but have no bearing on tracking statements, and secondly, to improve the accuracy of tracking change. Fig. 2 shows the summary generated for the program fragment on the left of Program 1. Given the two CFG summaries, DIRECT identifies sections of code that have been preserved using a text difference algorithm [16, 11, 4]. The text difference algorithm, given two text documents D and D', 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'.

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

- *Resource values.* In typical concurrent software, threads share resources and synchronize using concurrency control primitives that manipulate these re-

```
<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: The summary of the program on the left in Program 1

sources. Since DIRECT captures the control primitives and exposes them through states, the precise values of the resources can be accessed by the runtime manager. 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. Given a program state, the value of a resource r is represented by $val(r)$. We extract resource values whenever a state is observed. If the resource data is only manipulated at observable global states, then changes in the resource are observed with total precision.

- *Global variables.* Given a set G of global variables, we can include their values as part of the global state. Note that these values are not tracked whenever they change, but only when an observable statement is reached.
- *Extending observable statements.* Users can either expand on the set of OS primitives or library functions that DIRECT tracks or they may choose a program statement of interest, to be part of the set of observable statements.
- *Block execution times.* We track the average block execution times of each block in each thread. This is used to perform proportional delay injection. Whenever a new state is observed, we take the difference between the time at which the new state is observed and the time at which the previous state was observed for each thread.

4.2 Detecting New Events Efficiently

The use of DIRECT can be summarized as follows. First, a program P is run one or multiple times, generating a set of global states $E \subseteq \mathcal{E}$ encountered during these runs. Then, we exercise a modified version P' of P. P' is obtained either by source-code modifications from the user, or by the insertion of delays. Whenever during a run of P' a state e that is not present in E is encountered, DIRECT reports this new global state, along with (a suffix of) a sequence of global states leading to e.

For this technique to be practical, testing membership with respect to E must be performed efficiently both in terms of time and space. The time efficiency

is crucial for scalability to large programs. The space efficiency is especially important in the study of embedded software. The set E can be very large: for embedded software, we need an implementation that uses very limited amount of memory.

To achieve the desired efficiency, our implementation relies on Bloom filters for the representation of the set E. A Bloom filter is a probabilistically correct data-type that implements sets of objects, offering two operations: insertion and membership checking. A Bloom filter guarantees 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.

A Bloom filter maintains a table of M bits, initially all set to 0. It uses k hash functions, each mapping the encoding of an object into a number from 1 to M. An insertion consists of computing each of k hash functions over the object and setting to 1 the corresponding entries in the table. Checking membership consists of testing whether the bits at all position obtained by hashing the object are 1. If some entry is 0 then the object is reported as not being in the set. If all entries are 1 the object is declared to be in the set.

The accuracy of a Bloom filter does not depend on the size of the universe of objects or the encoding of each object. It depends only on the quality of the hash functions, and the number of different objects inserted. For k perfectly random hash functions, a table of M bits and n objects inserted, the probability of a false positive is:

$$(1 - e^{\frac{-kn}{M}})^k.$$

For example, the probability of a false positive in a Bloom filter with a table size eight times the number of elements inserted and five hash functions is typically less than 0.02 [9]. For example, a system with 625 global states can be tracked by a Bloom filter with less than 1 Kb of memory with very little ratio of false positives. A false positive in checking set membership using Bloom filters occurs because the hash functions may evaluate distinct global states to the same bucket. This property of Bloom filters implies that DIRECT may (rarely) miss new states. On the other hand, since there are no false negatives in checking set membership using Bloom filters, every new global state found by DIRECT is guaranteed to be new, in the sense that the reached state was not seen in the reference runs.

Each operation in the Bloom filter is also very efficient to perform. In our implementation, since independent hash functions with random properties are difficult to design, we use the technique of double-hashing [7] 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.

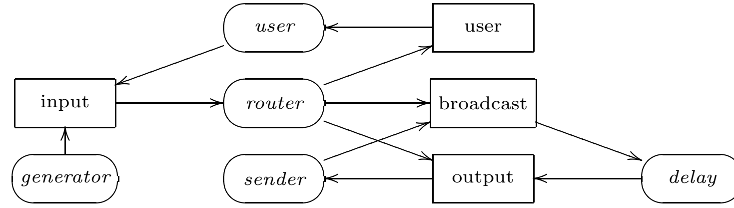


Fig. 3: Scheme of an ad-hoc network protocol implementation.

5 Case Studies

We tested the aforementioned techniques on two case studies. We compared solutions to the dining philosophers problem and analyzed an adhoc protocol for legOS, adapted to run in an eCos [1] environment.

5.1 An Adhoc Protocol

We analyzed a multi-threaded program that implements 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 in the figure.

Threads *user* and *generator* add packets to the *input* queue. The *router* thread 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, so they are consumed by the *user* thread. Packets in the *broadcast* queue are intended for broadcast, and they are moved 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 in Program 3. 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.

Finally, we place a check for the following invariant in the code,

$$val(., bb_free_sem) + val(., bb_els_sem) \in \{BB_NEL, BB_NEL - 1\}$$

where *BB_NEL* is the size of the broadcast queue. We modeled a plausible bug getting introduced when the blocking call to acquire the semaphore *bb_free_sem*

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);

52   // add an invariant check here.
53   semaphore_peek(&bb_free_sem, &freev);
54   semaphore_peek(&bb_els_sem, &elsv);
55   assert((freev + elsv) == BB_NEL || (freev + elsv) == (BB_NEL - 1));
     ...
```

is replaced 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. In Program 4 we show the snippet of code that incorporates this change. The bug introduced by this change is that the blocks of code that should execute when the semaphore is successfully acquired, terminate prematurely. Specifically, the call to post the semaphore *bb_els_sem* at line 51 in Program 4 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 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, leading to a violation of the invariant. 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 that is seen corresponds to the call 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 the trace difference in an html file. This file shows the first state in the test run that is not part of the set of states observed in the reference run. It also provides users with the ability to walk a suffix of the trace leading to the new global state. In the Appendix, we provide a more detailed explanation of the generated html files.

Program 4 A snippet of code from the packet router thread after changing a blocking call to be non-blocking. The bug is a consequence of premature termination of the trywait block.

```
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);

53  // add an invariant check here.
54  semaphore_peek(&bb_free_sem, &freev);
55  semaphore_peek(&bb_els_sem, &elsv);
56  assert((freev + elsv) == BB_NEL || (freev + elsv) == (BB_NEL - 1));
    ...
```

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
7	4	$(0,2)$	$(0,2)$	$(0,2)$	$(0,2)$	$(0,2)$	$(0,0,0,0,0)$	(4)	c
8	0	$(0,2)$	$(0,2)$	$(0,2)$	$(0,2)$	$(0,2)$	$(0,0,0,0,0)$	(0)	r
9	0	$(0,3)$	$(0,2)$	$(0,2)$	$(0,2)$	$(0,2)$	$(0,0,0,0,0)$	(0)	c
10	1	$(0,3)$	$(0,2)$	$(0,2)$	$(0,2)$	$(0,2)$	$(0,0,0,0,0)$	(1)	r
11	1	$(0,3)$	$(0,3)$	$(0,2)$	$(0,2)$	$(0,2)$	$(0,0,0,0,0)$	(1)	c
12	2	$(0,3)$	$(0,3)$	$(0,2)$	$(0,2)$	$(0,2)$	$(0,0,0,0,0)$	(2)	r
13	2	$(0,3)$	$(0,3)$	$(0,3)$	$(0,2)$	$(0,2)$	$(0,0,0,0,0)$	(2)	c
14	3	$(0,3)$	$(0,3)$	$(0,3)$	$(0,2)$	$(0,2)$	$(0,0,0,0,0)$	(3)	r
15	3	$(0,3)$	$(0,3)$	$(0,3)$	$(0,3)$	$(0,2)$	$(0,0,0,0,0)$	(3)	c
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

Table 1: A sequence of global states leading to a deadlock in a naive implementation of dining philosophers. Each state consists of the index of the active thread, all thread states, resource values, the set of resources held by the active thread and whether the event is a function call or return.

5.2 Dining Philosophers

We also analyzed a version of the classic dining philosophers problem using the Direct infrastructure. Program 5 shows the implementation of a philosopher.

Program 5 Dining philosopher

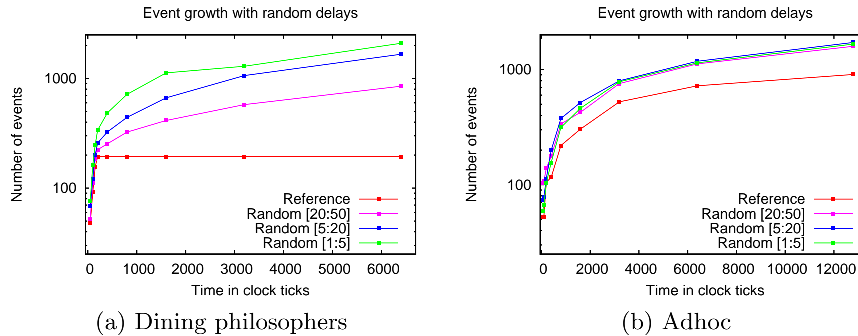
```
  void philosopher(int philosopher_id)
  {
    int first_fork, second_fork;

    // 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) {
      // thinking phase
0     thread_delay(20);

      // 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 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 of 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 second state in the table occurs when the fourth philosopher (thread T_3) has picked up her first fork, which corresponds to the return from the call in statement 1, after successful acquisition of the first resource. 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 monotone 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 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



(a) Dining philosophers (b) Adhoc

Fig. 4: Number of states observed as the running time increases, with and without injection of random delays, for the two case studies. The graphs show that using random delay injection the number of unique states seen increases, given the same test and target platform.

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? This question has a direct consequence on analyzing the impact of change in source code. 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 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.

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. The approach consists of instrumenting the program with a test harness and then exercising the instrumented program. Direct uses a combination of static analysis and runtime monitoring. The static analysis determines the instrumentation points, generates the monitoring code, and establishes the difference between two versions of a given program. The runtime instrumented code is executed before and after every concurrency primitive and user defined function. The runtime manager computes at each instrumentation point a global state consisting of the call stacks of every thread together with concurrency related data, and optionally some data selected by the user. The runtime manager collects the set of global states reached during the execution, and keeps a sequence of abstract global states leading to the current state.

For sensitivity analysis, the runtime manager inserts delays to simulate differences between platforms, libraries and operating systems. For change impact analysis, the runtime manager collects an over-approximation of the set of reached states of the original program. The states reached during the executions of the new version are then compared against the set of reached states of the original program. We presented two cases studies: an implementation of the so-called ad-hoc protocol for Lego robots, and an implementation of dining philosophers. 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. These case studies illustrate how the techniques described in this paper can help capture bugs in concurrency programs.

We 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 [10] we plan to proceed in several rounds, where the scheduler of the next round is obtained using static analysis and the set of global states obtained in the previous runs.

References

1. ecos homepage. http://ecos.sourceware.org/.
2. Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.
3. Johannes Bohnet, Stefan Voigt, and Jürgen Döllner. Projecting code changes onto execution traces to support localization of recently introduced bugs. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 438–442, New York, NY, USA, 2009. ACM.
4. Randal C. Burns and Darrell D.E. Long. A linear time, constant space differencing algorithm. In *Performance, Computing, and Communication Conference (IPCCC)*, pages 429–436. IEEE International, 1997.
5. Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Framework for testing multi-threaded java programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003.
6. Sunghun Kim, Jr. E. James Whitehead, and Yi Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, 2008.
7. Donald E. Knuth. *The Art of Computer Programming*, volume 3 (Sorting and Searching), chapter 6.4. Addison-Wesley, 2nd edition, 1998.
8. Thomas J. Marlowe and Barbara G. Ryder. An efficient hybrid algorithm for incremental data flow analysis. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 184–196, New York, NY, USA, 1990. ACM.
9. Michael Mitzenmacher and Eli Upfal. *Probability and Computing*. Cambridge University Press, 2005.
10. Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, pages 267–280, 2008.

11. Eugene W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.
12. George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Infrastructure for C program analysis and transformation. In *Proceedings of the 11th International Conference on Compiler Construction*, volume 2304 of *Lect. Notes in Comp. Sci.*, pages 213–228, 2002.
13. Xiaoxia Ren, Ophelia C. Chesley, and Barbara G. Ryder. Identifying failure causes in java programs: An application of change impact analysis. *IEEE Trans. Softw. Eng.*, 32(9):718–732, 2006.
14. Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: a tool for change impact analysis of java programs. *SIGPLAN Not.*, 39(10):432–448, 2004.
15. Barbara G. Ryder and Frank Tip. Change impact analysis for object-oriented programs. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 46–53, New York, NY, USA, 2001. ACM.
16. Walter F. Tichy. The string-to-string correction problem with block move. *ACM Trans. on Computer Systems*, 2(4), 1984.
17. Frank Tip. A survey of program slicing techniques. *J. Prog. Lang.*, 3(3), 1995.

A Appendix

We now include screen-shots of the html that is generated by Direct, when a new event is reached in a run due to source code change. These figures show the new event when we replace blocking semaphore requests with non-blocking requests in the code implementing the adhoc protocol. The new event is exactly the point at which the newly introduced code causes a bug. We used Direct to run a single test on the program before and after the change, storing the sequence of events observed in each run explicitly. We then used a *compare* feature of Direct that compares two sequences of events to extract the matching prefix of events into data files that can be navigated using an html driver. The html driver has buttons to navigate the sequence of events in the second run (after code change) that lead to the new event: an event not present in the sequence of events observed in the first run (before code change). Figs. 5 and 6 show the last two steps in this sequence. On the right hand side in both figures is the set of thread states in the first run, at the point at which Direct found a new event in the second run. The lines marked with a lighter background point to the last thread state for each thread. In the execution model of interleaving semantics that we consider, it is clear that each new event has exactly one thread that witnesses a new thread state. The thread state corresponding to this thread is marked with a darker background.

```
packet_router

518   #endif
519         // First, we check the nonce, to avoid a broadcast storm.
520         // If the nonce is not new, we do nothing.
521         if (nonce_is_new (in_pkt->nonce)) {
522   #ifdef DEBUG
523    diag_printf ("Nonce is new\n"); // debug
524   #endif
525    saw_nonce (in_pkt->nonce);
526    if (in_pkt->dst == local_address) {
527      // It's for us.
528      if (cyg_semaphore_trywait (&bb_free_sem))

529            cyg_semaphore_wait (&bb_mutex);
530
531   #ifdef DEBUG
532            diag_printf ("It is for me, and I got space for RP in the out buffer\n"); //
533   #endif
534            out_buf = bb_buf [bb_in];
535            out_pkt = (rap_pkt_t *) out_buf;
536            // If we are the destination, then we need to reply with an RP packet.
537            out_pkt->len      = in_pkt->len + 3; // There is the new reverse list at the
538            out_pkt->ptype    = PTYPE_RP;

Thread: packet_send [./adhoc.c]

packet_send

1075      /*      diag_printf ("Sending out a packet: \n");  */
1076      /*    } */
1077      /*    print_packet (in_buf);  */
1078      /*    diag_printf ("================\n");  */
1079    #else
1080       result = lnp_integrity_write (in_buf, len);
1081    #endif
1082       if (result) {
1083         // If sending is not successful, puts the buffer back into the BB
1084         // buffer, to do a primitive kind of random delay.
1085         cyg_semaphore_wait (&bb_free_sem);

1086         cyg_semaphore_wait (&bb_mutex);
1087         memcpy (bb_buf [bb_in], in_buf, len);
1088         bb_in = (bb_in + 1) % BB_NEL;
```

```
packet_router

518   #endif
519         // First, we check the nonce, to avoid a broadcast storm.
520         // If the nonce is not new, we do nothing.
521         if (nonce_is_new (in_pkt->nonce)) {
522   #ifdef DEBUG
523    diag_printf ("Nonce is new\n"); // debug
524   #endif
525    saw_nonce (in_pkt->nonce);
526    if (in_pkt->dst == local_address) {
527      // It's for us.
528      cyg_semaphore_wait (&bb_free_sem);

529      cyg_semaphore_wait (&bb_mutex);
530
531   #ifdef DEBUG
532      diag_printf ("It is for me, and I got space for RP in the out buffer\n"); // debug
533   #endif
534      out_buf = bb_buf [bb_in];
535      out_pkt = (rap_pkt_t *) out_buf;
536      // If we are the destination, then we need to reply with an RP packet.
537      out_pkt->len      = in_pkt->len + 3; // There is the new reverse list at the end
538      out_pkt->ptype    = PTYPE_RP;

Thread: packet_send [/tmp/ref/adhoc.c]

packet_send

1074      /*      diag_printf ("Sending out a packet: \n");  */
1075      /*    } */
1076      /*    print_packet (in_buf);  */
1077      /*    diag_printf ("================\n");  */
1078    #else
1079       result = lnp_integrity_write (in_buf, len);
1080    #endif
1081       if (result) {
1082         // If sending is not successful, puts the buffer back into the BB
1083         // buffer, to do a primitive kind of random delay.
1084         cyg_semaphore_wait (&bb_free_sem);

1085         cyg_semaphore_wait (&bb_mutex);
1086         memcpy (bb_buf [bb_in], in_buf, len);
1087         bb_in = (bb_in + 1) % BB_NEL;

First  Prev  Next  Last
```

Fig. 5: Figure that shows the penultimate event in the sequence leading up to a new event after code change. The last thread state for each thread is shown with a lighter background. The state of the active thread is shown with a darker background.

packet_router

```
545              memcpy (out_pkt->sr.data, in_pkt->sr.data, in_pkt->sr.len);
546              out_pkt->sr.data [in_pkt->sr.len] = local_address;
547              // Creates the reverse list; the +1 is for the length of the in_pkt->sr.len
548              out_rev_list = (rap_list_t *) (out_buf + HEADER_LEN + out_pkt->sr.len + 1);
549              out_rev_list->len = 1;
550              out_rev_list->data [0] = local_address;
551              // Done.
552              bb_in = (bb_in + 1) % BB_NEL;
553              cyg_semaphore_post (&bb_mutex);
554            }
555            cyg_semaphore_post (&bb_els_sem);

556
557            // add an invariant check here.
558            cyg_semaphore_peek (&bb_free_sem, &freev);
559            cyg_semaphore_peek (&bb_els_sem, &elsv);
560            cyg_semaphore_peek (&bb_mutex, &mutexv);
561
562            if ((freev + elsv) != BB_NEL &&
563                (freev + elsv) != (BB_NEL - 1)) {
564              diag_printf("Invariant violation! bb_free_sem = %d, bb_els_sem = %d\n",
565                          freev, elsv);
```

Thread: packet_send [./adhoc.c]

packet_send

```
1075     /*      diag_printf ("Sending out a packet: \n");  */
1076     /*    } */
1077     /*    print_packet (in_buf);  */
1078     /*    diag_printf ("================\n");  */
1079   #else
1080       result = lnp_integrity_write (in_buf, len);
1081   #endif
1082       if (result) {
1083         // If sending is not successful, puts the buffer back into the BB
1084         // buffer, to do a primitive kind of random delay.
1085         cyg_semaphore_wait (&bb_free_sem);

1086         cyg_semaphore_wait (&bb_mutex);
1087         memcpy (bb_buf [bb_in], in_buf, len);
1088         bb_in = (bb_in + 1) % BB_NEL;
```

packet_router

```
518   #endif
519         // First, we check the nonce, to avoid a broadcast storm.
520         // If the nonce is not new, we do nothing.
521         if (nonce_is_new (in_pkt->nonce)) {
522   #ifdef DEBUG
523    diag_printf ("Nonce is new\n"); // debug
524   #endif
525    saw_nonce (in_pkt->nonce);
526    if (in_pkt->dst == local_address) {
527      // It's for us.
528      cyg_semaphore_wait (&bb_free_sem);

529      cyg_semaphore_wait (&bb_mutex);
530
531   #ifdef DEBUG
532      diag_printf ("It is for me, and I got space for RP in the out buffer\n"); // debug
533   #endif
534      out_buf = bb_buf [bb_in];
535      out_pkt = (rap_pkt_t *) out_buf;
536      // If we are the destination, then we need to reply with an RP packet.
537      out_pkt->len     = in_pkt->len + 3; // There is the new reverse list at the end
538      out_pkt->ptype   = PTYPE_RP;
```

Thread: packet_send [/tmp/ref/adhoc.c]

packet_send

```
1074     /*      diag_printf ("Sending out a packet: \n");  */
1075     /*    } */
1076     /*    print_packet (in_buf);  */
1077     /*    diag_printf ("================\n");  */
1078   #else
1079       result = lnp_integrity_write (in_buf, len);
1080   #endif
1081       if (result) {
1082         // If sending is not successful, puts the buffer back into the BB
1083         // buffer, to do a primitive kind of random delay.
1084         cyg_semaphore_wait (&bb_free_sem);

1085         cyg_semaphore_wait (&bb_mutex);
1086         memcpy (bb_buf [bb_in], in_buf, len);
1087         bb_in = (bb_in + 1) % BB_NEL;
```

First | Prev | Next | Last

Fig. 6: The new program event after the code change.