Closing the Programmer’s Universe: A Pattern Language for Reproducibility in Concurrent Programming Environments

Douglas Niehaus and Jerry James
{niehaus,james}@eecs.ku.edu
Electrical Engineering & Computer Science Department
Information and Telecommunication Technology Center
University of Kansas, Lawrence

Christopher Gill
cdgill@cse.wustl.edu
Department of Computer Science and Engineering
Washington University, St.Louis

Abstract
Application software implemented as a set of concurrent computations has become a vital feature of an ever-widening range of systems. However, concurrent programming carries a latent danger of incorrect behavior due to undetected errors in concurrency control. This danger is particularly subtle because such incorrect behavior is typically intermittent and thus essentially irreproducible.

This paper presents a pattern language for addressing reproducibility of concurrent computation execution. It resolves specific design forces of concurrent programming environments by establishing a framework of patterns within which such reproducibility is a first class design criterion. This pattern language generates designs in which reproducible concurrent execution can be achieved, thus providing a necessary foundation for scientifically valid and efficient experimentation with concurrent applications.

Keywords: Concurrency, Concurrency Control, Distributed Systems, Middleware, Operating Systems, Reproducibility, Debugging, Testing.

1 Introduction
Application software implemented as a set of concurrent computations has become a vital feature of an ever-widening range of activities, from the embedded control systems in automobiles to transactions in financial markets. Applications executing within concurrent programming environments carry with them a latent danger of incorrect behavior due to undetected errors in their concurrency control. This danger is particularly subtle because such incorrect behavior is typically intermittent and thus essentially irreproducible. This makes it difficult to be certain a problem actually exists and usually makes it arduous to diagnose and correct its cause. Reproducible experiments are fundamental to the scientific method of problem solving. Without reproducibility we cannot conduct scientifically valid debugging or testing of concurrent software, thus significantly restricting the level of assurance that testing and debugging can provide.

This paper presents a pattern language addressing reproducibility of concurrent computation execution in the context of pseudo-concurrent multi-threaded systems. This approach has also been applied to concurrency arising from distribution, but that is beyond the scope of this paper due to length constraints. The pattern language presented here resolves specific design forces of current programming environments by (1) establishing a framework within which reproducibility is a first class design criterion, and (2) generating designs in which it can be achieved. This pattern language thus provides a necessary foundation for scientifically valid and efficient testing and debugging of concurrent applications.

This section first introduces some basic terms, then discusses reasons that many concurrency control errors are intermittent, and finally provides an overview of how our approach supports reproducibility. The pattern language we present must solve three problems to achieve reproducibility. First, it must enable the developer to record the sequence of events that occurs during a particular execution of a computation. Second, it must permit the developer to execute a specific sequence of events. Third, it

---

1In a multi-threaded system with physical concurrency, some CPU instructions are non-atomic. Dealing with this problem is beyond the scope of this paper.
must allow developers to construct event sequences that have not yet been recorded during a test. We provide an overview of these issues in Section 1.3, before covering them in detail in the rest of the paper.

1.1 Terminology

Supporting reproducibility of concurrent computation behavior requires nothing more, but also nothing less, than: (1) the ability to record computation behavior in a given execution instance and (2) the ability to control a subsequent execution of the computation to produce the same behavior. Three factors make the design and implementation of programming environments providing the reproducibility property extremely difficult. First, because execution behavior of concurrent computations is determined by the interactions of a number of influences, whose interactions are sometimes subtle. Once identified, methods for recording each occurrence of every influence and forcing it occurrence during reproduction must be established. Second, because the set of influences on concurrency exist in many different parts of a computer system. Third, because reproducibility is an all or nothing property. If a programming environment fails to adequately record and control even a single influence on concurrent behavior, then the environment is extremely unlikely to support concurrency reproducibly in a useful way.

Events influencing the concurrent behavior of computations, and which are thus relevant to reproducibility, include: execution of CPU instructions, signal delivery and handler execution, I/O operations, and system calls. The discussion of these topics at the level of detail required to explain the pattern language and its uses involves fairly large and unusual combination of concepts. We assume for the moment that this set of events is sufficient to support reproducibility for the classes of concurrent computations discussed in this paper. We now introduce a set of terms that will be required in the rest of the paper which presents the pattern language, and shows that the set of events in indeed adequate to support reproducibly for the classes of computations discussed in this paper.

Given that we know the complete set of relevant events, a thread event history is then the totally ordered sequence of events experienced by a single thread of execution. At the most literal level we could record the execution of each instruction, but as discussed in Sections 3.1.1 and 3.2.1, it turns out to be adequate to record the entry of the thread into each basic block of the program. The term “basic block” originates in the technical literature of language compilation and compiler implementation. The term refers to a segment of machine code produced by the compiler which is executed linearly; without a branch instruction, except perhaps at the end. Basic blocks are produced for some simple program statements in the source language, but many source program statements are translated into multiple basic blocks.

Signals are a feature of many but not all programming environments, and are essentially independent of the programming language used. In general terms, a signal is an asynchronous event that can happen at any time during the execution of the computation. Stated another way, signals can be delivered between any two machine instructions executed by the computation. Many systems define a wide range of signals, and provide the ability to associate a handler with each signal. A signal handler is implemented by the user as simple subroutine. The invocation of the handler is the unusual aspects of signals.

When a signal is delivered to a thread, the operating system controlling the computation arranges for the execution of the handler in the current context of the computation. The effect of the this on the event history is to make it appear as if a call to the handler subroutine existed in the computation code at the point defined by the computation’s current position in its event history. While no such subroutine call exits, the event history includes the instruction execution events of the handler as if it did, including the return from the handler.

For a computation implemented using a single thread, the computation event history is identical to the thread’s event history. For a computation implemented using multiple threads, the computation event history is the sequence resulting from the interleaving of segments of thread event histories by the set of context switches in a particular execution of the computation. We note that all events are viewed as being delivered to a particular thread. Therefore, context switches are in fact pairs events one noting the switch away event that occurs in the previously executing thread’s history, and the switch to that occurs in the newly executing thread’s history. These paired context switch events thus delimit points of transition between segments of different thread histories under a particular interleaving.
1.2 Variable Behavior

Errors due to incorrect concurrency control are essentially irreproducible because current programming environments do not provide sufficient control over application execution to produce a specific computation event history. Thus, even if a developer has hypothesized that a particular set of computation histories would exhibit a specific problem, most programming environments provide no way to effectively test that hypothesis.

Concurrent computation behavior is affected by a number of influences that can be categorized in several ways. Some influences are fundamental in the sense that they are aspects of the application environment, and remain constant across variations in implementation. Examples of fundamental influences include: interactions with users through interface devices, arrival of signals from sensors, variation in content of input files, and arrival of network messages. Contextual sources of variation are influences that lie within the program’s execution environment, but which lie outside the programming model available to the developer. Examples include system thread scheduling decisions, resource allocation operations, signal delivery for programming environment reasons, and system state information (such as CPU load) that some computations use as input information, and which can thus affect computation execution behavior.

Computation behavior is fundamentally determined by control flow within a thread and context switching between threads. Influences on behavior can also be distinguished by whether they exert their influence on context switch and control flow behavior directly or indirectly. An example of a direct influence on control flow is the delivery of a signal, which essentially executes a call to a subroutine that is not present in the current context, at the current program counter value. On the other hand, data read through I/O operations which affect evaluation of conditional statements are an indirect influence on control flow. An example of a direct influence on context switching is a scheduling decision invoked by a timer interrupt which decides to perform a context switch. A system call requesting I/O that causes the calling thread to block, thus provoking the scheduler to context switch to another thread, is an example of an indirect influence on context switching.

The previous examples are consistent with a set of largely independent threads reacting to events outside the computation and performing I/O with the external environment. Many computations are also implemented as sets of threads that interact through a set of shared data. Concurrency control elements of the program environment, e.g., semaphores and condition variables, exist to enable developers to constrain the set of possible computation event sequence for purposes of ensuring correct behavior. While concurrency control functions in the programming model can be used in a variety of ways, the most common is constraining access to shared data to ensure its consistency.

The influence of concurrency control on computation behavior is contextual because the definition of the sets of shared data is an aspect of software design, and could vary significantly among different implementations of the same computation. The influence of concurrency control on computation execution is indirect because threads block on access to a set of shared data that is already being used, thus inducing a context switch to a runnable thread.

Errors due to incorrect concurrency control usually occur only within a small subset of all possible computation histories. The specific history produced by a particular computation execution is subject to a number of direct and indirect influences that are both fundamental and contextual, as well as being both synchronous and asynchronous (Force F4) with the computation. From the user’s point of view, under commonly used program execution environments, the specific history produced is thus subject to a number of apparently randomizing factors.

Errors due to incorrect concurrency control are thus irreproducible within these environments because there is no way for the developer to guarantee that a computation will follow a specific history producing the problem behavior of interest. The pattern language described in this paper adds reproducibility to the program execution environment by providing control over the full set of influences affecting the event history produced by a computation.

1.3 Supporting Reproducibility

Any programming environment or software architecture wishing to support reproducible concurrency must satisfy three basic requirements:
1. provide the ability to record all computation events that matter in reproducing concurrent behavior,

2. provide the ability to control the occurrence of these relevant events, and

3. provide a framework and tools using the control capability to enable interactive and automated guiding of a computation through a specific history

Applications and program execution environments can satisfy these requirements in a variety of ways, depending on the detailed context and features of the concurrent programming model being used. For testing and debugging purposes, developers will often benefit from the ability to synthesize event histories representing scenarios of interest. This and other design forces that must be resolved by the pattern language are discussed in Section 2. Section 3 presents the patterns in the language and offers a narrative for how those patterns can be composed to generate a coherent design. Section 4 describes other work related to the patterns in this paper. Finally, Section 5 offers conclusions and describes future work.

2 The Forces

The design forces addressed by this pattern language are summarized in Table 1, in the order in which they are discussed in this section. Table 1 gives a label and name for each force, the consequences of the force, and labels for the patterns described in Section 3 that address each force. The pattern language described in this paper is motivated by the desire to characterize software behavior and the corresponding need for reproducibility of concurrent computation behavior. Behavioral characterization and reproducibility are the first two design forces the pattern language addresses. The other design forces addressed are implications of the needs to characterize and reproduce behavior, and of the mechanisms used to support them in a programming environment.

F1: Characterizing the Behavior of Concurrent Software is Hard Developers often need to characterize the behavior of their software as part of the development process, but also as part of assurance testing. Exploring the full range of possible concurrent behaviors is difficult by definition, but it is impossible without certain kinds of instrumentation and control in the programming environment.

F2: Reproducibility is Necessary for the Scientific Method Reproducibility of a specific computation behavior is fundamental to systematic investigation of a hypothesis, which attempts to explain observed behavior, using the scientific method. The ability to reproduce specific computation behavior allows a developer to conduct scientifically valid experiments during design, testing and debugging. Reproducibility is thus a fundamental aspect of effective support for development of concurrent software.

F3: Computation Behavior is Variable Fine grain behavior of a computation that is multi-threaded or distributed can be affected by a number of factors. The combination of these influences makes the fine grain behavior of one instance of a computation appear to vary in random ways from the behavior of another instance, as was discussed in Section 1.2.

F4: Reproducibility Requires Detection of a Complete Set of Events The complete set of events required to achieve reproducibility are not usually available in current programming environments, although many provide a significant subset. Relevant events are those that must be known to ensure correctness when reproducing the execution of a multi-threaded computation. Some events of interest are completely synchronous with a thread and its virtual machine, such as the ordering of basic block execution or the reading of data from a file. Asynchronous events of interest occur at arbitrary points in the sequence of synchronous thread events, since they arise from outside the computation's virtual machine. An example of such an asynchronous external event is delivery of a signal notifying the thread of an external world event occurrence.

F5: Single Events Do Not Completely Describe Computation Behavior Reproduction of concurrent behavior must accurately relate the sequence of relevant computation events; we call this the history of the computation. The executed instructions of the several threads implementing a computation are the most obvious elements of a complete history, but information from the programming environment should also be part of a complete com-
Table 1: Summary of Reproducibility Design Forces

<table>
<thead>
<tr>
<th>#</th>
<th>Force</th>
<th>Consequences</th>
<th>Relevant Pattern(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>Characterizing Software Behavior</td>
<td>Desire for Examination</td>
<td>P1, P2, P3, P4</td>
</tr>
<tr>
<td>F2</td>
<td>Need for Reproducibility</td>
<td>Valid Experimentation</td>
<td>P1, P2</td>
</tr>
<tr>
<td>F3</td>
<td>Variable Computation Behavior</td>
<td>Challenge for Reproducibility</td>
<td>P1, P2</td>
</tr>
<tr>
<td>F4</td>
<td>Observing Complete Set of Events</td>
<td>Support Reproduction</td>
<td>P1</td>
</tr>
<tr>
<td>F5</td>
<td>Single Events do not Describe Behavior</td>
<td>Need to Collect a Sequence</td>
<td>P1</td>
</tr>
<tr>
<td>F6</td>
<td>Histories of Interest</td>
<td>Synthesis Desired</td>
<td>P2, P3, P4</td>
</tr>
<tr>
<td>F7</td>
<td>Arbitrary History Execution</td>
<td>Produce Specific Behavior</td>
<td>P2</td>
</tr>
<tr>
<td>F8</td>
<td>Assurance Requires Covering the Full Range of Behaviors</td>
<td>Complete Testing</td>
<td>P3, P4</td>
</tr>
</tbody>
</table>

Computation history. For example, the point during thread execution at which a signal is delivered, the content of the computation’s input/output data streams, and the points in each thread’s instruction sequence at which the CPU is switched from executing one thread to another are all necessary elements of a complete computation history.

**F6: Histories of Interest May Not Occur During Observed Executions**  Effective testing and debugging often involves the generation of a hypothesis by a developer about how a given behavior of interest might occur, and then investigation of that hypothesis by experiment. However, the variation in computation behavior, Force F3 above, means that a developer can neither assume that a behavior of interest will appear in a given experiment, nor that the behavior in question will appear at all during any period of observation. This is important because many errors in concurrency control are intermittent since they depend on the coincidence of several independently varying influences.

**F7: Arbitrary History Execution is Required for Investigation**  Hypothesis generation and testing is an important aspect of concurrent software design, development, debugging, and testing. In this context, it is clear that developers require the ability to execute a particular scenario of interest. A given execution scenario is produced by a set of one or more computation execution histories. There are several reasons a user may want to execute a specific history. The most obvious is to reproduce recorded behavior during an execution exhibiting a behavior of interest. However a user may also want to execute a synthesized history to test an hypothesis about a possible behavior that has not yet been observed or recorded. In either case, the programming environment must provide tools and support that enable the user to follow a specific execution event history.

**F8: Assurance Requires Covering the Full Range of Behaviors**  Software can exhibit a wide range of behaviors in many cases, but however complex the space of possible behaviors, assurance requires covering the full range. Anything less leaves a portion of the behavior space unexamined.

3 The Patterns

The patterns described in this section resolve the forces described in Section 2. The essential elements of this resolution include the ability to: (1) observe all relevant events in the computation and in its execution environment, (2) control a computation’s execution to follow a specific event history, (3) synthesize a history of interest that may not yet have been observed during execution, and (4) cover a set of equivalence classes of execution event histories with a set of representative histories. Table 2 summarizes the patterns in this language, giving a label and name for each pattern, and the design forces it resolves.

The patterns in the language are described in Section 3.1. Section 3.2 then presents a narrative description of a use of the pattern language to implement a program development and execution environment featuring user-level multi-threading, computation event history recording, and event history execution.
Table 2: Summary of Reproducibility Patterns

<table>
<thead>
<tr>
<th>#</th>
<th>Pattern</th>
<th>Forces Resolved</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Complete Event Sequence Recording (CESR)</td>
<td>F1, F2, F3, F4, F5</td>
</tr>
<tr>
<td>P2</td>
<td>Event Sequence Execution (ESE)</td>
<td>F1, F2, F3, F6, F7</td>
</tr>
<tr>
<td>P3</td>
<td>Event Sequence Synthesis (ESS)</td>
<td>F1, F6, F8</td>
</tr>
<tr>
<td>P4</td>
<td>Event Sequence Coverage (ESC)</td>
<td>F1, F6, F8</td>
</tr>
</tbody>
</table>

3.1 Pattern Overview

This section presents the four patterns in the language. The first three address recording, reproducing, and synthesizing computation event histories, which form the basis for reproducing and scientifically valid experiments. The fourth pattern addresses the issue of coverage. In this context, the term coverage refers to how well a given set of histories covers the set of all classes of behavior. Here, the classes of behavior are defined by an analysis of the code with respect to a behavioral property.

3.1.1 Pattern P1: CESR

Name: Complete Event Sequence Recording (CESR)

Problem: Software development, testing, and debugging involves a significant aspect of learning about computation software structure and the range of its execution behavior. An essential element of this learning process is the correlation of a sequence of events during the computation’s execution with its behavior. Observing such a correlation obviously depends on the ability to know what sequence of events occurred during a given execution of the computation. The set of events recorded must be complete in the sense that it includes every event of interest. In this context, an event of interest is any event that must be known to support reproducible execution of a concurrent computation. Clearly, while observing any event of interest is desirable, observing every event of interest is a different and higher standard of recording. How much should we record so that the behavior of a computation can be understood and reproduced?

Forces: Characterizing the behavior of concurrent software is hard (F1), and computation behavior is variable (F3). However, reproducibility is necessary for the scientific method (F2). Reproducibility requires detection of a complete set of events (F4), but single events do not completely describe computation behavior (F5).

Context: Design, development, testing, and debugging of multi-threaded or distributed applications based on modern operating system semantics, such as those specified by POSIX.

Solution: Record all events that are important with respect to behaviors of concern to the user. In the context of this pattern language, these are the set of events that can affect concurrent computation behavior under the programming model in use. As Figure 1 illustrates, there are several types of events that come from multiple sources, and the set of events required is a function of the programming model. Examples of specific events that are relevant for ensuring the reproducibility of concurrent program execution behavior under some familiar programming models include:

1. machine instructions (basic blocks)
2. context switches
3. signals
4. I/O
5. semaphore operations
6. monitor operations
7. condition variable operations
8. Ada rendezvous

Figure 1 illustrate four types of computation events. The most obvious is the execution of a basic block of code by a thread. Each basic block of a program is uniquely labeled. Note that in the figure the events indicate both which basic block is being executed and the thread executing it. Context switches can happen between any two machine instructions during execution, although for simplicity the figure illustrates a context switch between threads $P$ and $Q$ at basic block boundaries. Signals can also happen between any two instructions, and the figure illustrates this by showing the execution of basic block
being interrupted by a signal, whose sequence of basic blocks are shaded Grey. Basic block $P3$ is thus split into the $P3a$ and $P3b$ portions. Finally, note that the I/O operation performed by thread $Q$ causes a context switch back to thread $P$, which would be a common though not inevitable occurrence.

**Consequences:** A non-trivial set of events must be recorded to support reproducibility of computation execution, and long sequences of these events may be required to describe a given execution. Thus, gathering a complete execution event sequence can produce very large data sets. It also requires instrumentation of both the computation and its execution environment to gain access to all relevant events. A more subtle issue is the completeness of the event set, with respect to a given kind of analysis. In this pattern language the relevant analysis considers the reproducibility of concurrent execution behavior within a specific program execution environment, which defines the set of events that we need to observe and record.

**Known Uses:** Examples of this problem and solution can be found in a wide variety of testing and debugging situations. The most obvious example is that nearly all developers instrument their code and collect a sequence of some set of events from time to time, often using simple `print` statements. It is important to note that while many of these examples enable the collection of various events, the sets of events collected are generally not complete with respect to the property of reproducibility which is a focus of this paper.

Instrumenting compilers address the problem of event recording in a slightly different form, producing code during compilation that records the entry into each basic block during execution. In the case of instrumentation to measure test suite code coverage, these compilers add code to each basic block noting whether it has been executed during a given program execution and then accumulate that information across all the executions in a test suite. Instrumentation for performance profiling also adds code to each basic block, but in this case the code increments counters to support construction of a statistical profile showing how much of the total execution time is spent in each section of the program.

The programming environment which we have developed at KU provides a similar compiler instrumentation feature to record a significant set of computation events during execution. In this case, the instrumentation added to each basic block places an entry in the computation history noting entry to the block. While not immediately obvious, this turns out to be sufficient to know when in the history each machine instruction was executed. Note, however, that the program execution environment must also add certain events to the recorded history. In the case of our current reproducible environment BThreads [15], these include thread context switches and signal delivery.

Most programming environments include a debugger, which provides the developer with a controlled environment within which to observe computation execution. Examples of debuggers that enable users to observe relevant types of events include GDB, WindView, and DBX, distributed debugging environments depending on history replay, and the Java omniscient debugger [12].
3.1.2 Pattern P2: ESE

**Name:** Event Sequence Execution (ESE)

**Problem:** Given that we have collected a complete sequence of events that occurred during a given execution of a computation, we must now address the reproduction of that sequence in a subsequent execution. This is necessary because unless a specified sequence can be executed, the fundamental goals of characterizing software behavior and conducting scientifically valid experiments cannot be realized.

**Forces:** Characterizing the behavior of concurrent software is hard (F1), and computation behavior is variable (F3). However, reproducibility is necessary for the scientific method (F2). Histories of interest may not occur during observed executions (F6), and execution of arbitrary histories is often required for investigation (F7).

**Context:** An environment within which we can (1) record an event history including all events that are necessary to fully reproduce an execution scenario (P1:CESR) and (2) reproduce the recorded execution history. Beyond that, the solution described below is applicable to any software whose behavior is of interest to a user, who wishes to consider a specific execution scenario.

**Solution:** The problem can be solved by using the input history, recorded or synthesized, to guide computation execution under the control of a debugger, as Figure 2 illustrates. Clearly, this debugger requires a slightly larger set of capabilities than those typically present. The new capabilities include the ability to follow the input history, comparing it to the history generated by the computation whose execution is being guided. As the debugger guides the computation execution it compares the history produced by the computation in its current context to the input history to determine where in the current computation’s sequence event guidance will next be required. The guiding debugger can set a breakpoint at the proper place in the code, and then wait for the computation to reach that place in the proper context. When it does, the guiding debugger causes the event specified by the input history to occur. In the situation provided by the figure, the guiding debugger would place a breakpoint in the code where the I/O takes place, and then force a context switch from thread Q to thread P.

This pattern resolves Force F7 by addressing the execution of an arbitrary input history, and it contributes to the resolution of Force F6 by enabling a user to monitor execution of a synthesized history of interest during the process of construction to ensure its feasibility. This pattern completes the resolution of Force F3 by providing an environment within which the tool guiding computation execution can control event occurrence, thus eliminating variability in computation behavior in the guided context. Most important, in cooperation with pattern P1:CESR, this pattern resolves Force F2, since it enables users to reproduce any desired computation behavior for further study. This pattern also helps with the resolution of Force F1 since it supplies a useful ability for users wishing to characterize the behavior of a given computation.

**Consequences:** Users can observe computation behavior under a given history step by step, examining causes and testing the efficacy of solutions to specific problems.

**Known Uses:** The vast majority of program development environments include some kind of debugging support which enables the developer to guide an execution of their program. However, these systems typically do not provide control over the full range of behaviors required to reproduce the behavior of a concurrent computation from a previous run, nor do they include the capability to follow an input computation event history.

Most developers using current debugging environments implement a restricted instance of this pattern by hand when they set up specific execution scenarios and control various aspects of program behavior using debugger
commands. The Java omniscient debugger [12] also provides a partial instance of the pattern, due to its ability to show the state of the program at arbitrary points in its execution. However, it can only show the state of an actual execution; it does not include the capability of following an input computation history.

A previous project at KU, SmartGDB, extended the GNU debugger GDB to address two drawbacks it had at that time: limited scripting capabilities, and limited support for controlling multi-threaded computations. The SmartGDB project integrated the Tcl/Tk scripting and GUI support framework with the command language of GDB, producing a hybrid scripting language within which it was possible to attach procedures containing GDB commands to breakpoints [8]. SmartGDB also contained an interface to a simple API in a threading package that explored the use of guided execution scenarios for multi-threaded programs by permitting the controller to force a context switch among threads at a breakpoint. SmartGDB thus also provided a partial implementation of this pattern in that the user could produce any desired execution history under manual control, but it did not provide the ability to automatically follow an input history.

Debuggers addressing distributed systems which include the ability to replay a recorded history of events for the distributed computation come close to fully implementing this pattern, although they generally concentrate on histories tracking only messages, and not context switches or the delivery of signals in multi-threaded computations on single nodes.

The prototype user-level multi-threaded program development and execution environment we have developed, which is discussed in Section 3.2, is the system among those of which we are aware that goes farthest in implementing this pattern. The Clever Insight debugger is capable of following an input history, and of monitoring the progress of the executing computation to identify points at which context switches and signal delivery should occur [13]. It is also able to use the control interface provided by the BThreads library [15] to cause the required context switch or signal delivery. It is important to note that even this project as yet fully implements this solution only for compute bound computations with signals. Work is currently being done to extend the capabilities of Clever Insight and BThreads to include I/O operations and system calls through encapsulation and I/O stream capture and replay.

3.1.3 Pattern P3: ESS

Name: Event Sequence Synthesis (ESS)

Problem: The user can imagine a plausible and interesting execution history of a program but is unable to create an conventional test that will produce the desired execution history due to uncontrolled factors such as asynchronous events including context switches and signal delivery timing. The user thus needs a way to create a possible event history that may not occur during conventional execution during testing.

Forces: Characterizing the behavior of concurrent software is hard (F1). Histories of interest may not occur during observed executions (F6). However, assurance requires covering the full range of relevant behaviors (F8).

Context: This problem can arise in several ways. For example, developers are often faced with a situation where either they, or a user, have observed an interesting computation behavior, but they are unsure of why it occurred and they are unable to reliably reproduce it during an instrumented execution of the computation. In a related situation, a developer may imagine a particular sequence of events during execution that is of interest for one reason or another, and wish to examine the implications of the scenario more closely.

In these and other cases, such as test suite design or during development, a programmer will form hypotheses about how the behavior of interest might be produced. Sometimes the possible causes of the behavior are obvious, and if it represents a problem, its solution may also be obvious. Often, however, the possible causes of the behavior are subtle, depending on the convergence of sets of events and conditions that vary significantly from run to run.

In such cases, developers often must hypothesize a fairly complex sequence of events to explain the behavior, and the chances that a given execution of the computation will experience all of the required events and conditions in the correct order is vanishingly small. In these cases, the user would find the ability to synthesize an hypothesized execution history extremely useful.
Solution: Solving this problem requires the creation of a way for users to synthesize possible computation event histories that exhibit properties of interest. While a user might be interested in a specific unique history, more often a user is interested in examining a representative of an equivalence class. In other words, the set of specified constraints can be satisfied by more than one possible history. The key point is, however, that a synthesized history must not merely be plausible, it must be possible.

A number of approaches are possible. The simplest is to have the user write a history, but this method suffers the significant limitation that it is easy for a human to imagine an impossible execution history. Other approaches could include techniques that resemble some aspects of model checking and partial evaluation. The method we use in our programming environment investigating reproducibility is construction by guided execution. Under guided execution, the user specifies a set of way points at which the events affecting concurrent behavior occur.

Consequences: Synthesized sequences must be checked for feasibility, and the methods used to automate checking of a synthesized history against the behavior it induces must be able to avoid the \textit{halting problem}. This in turn requires intervention by a human agent, a timeout, or use of some other mechanism capable of ensuring termination of the checking computation itself.

Known Uses: Developers provide a partial implementation of this pattern when they set up a specific execution scenario, but they typically lack sufficient control over the computation and its environment to force a specific execution history to occur. Further, all developers of concurrent know that it is possible to confidently imagine an execution history on one day that analysis on another day shows is impossible. It is clear than an effective solution to this problem must include strong support for checking that a synthesized history is possible.

While the synthesis of a specific computation history can be approached in a number of ways, in our environment we use a simple but effective combination of specification by the user and feasibility checking by the support tools. We call this approach \textit{synthesis by construction} using \textit{guided execution}. Figure 3 illustrates our approach.

In this approach, the user guides execution of the computation under control of the Clever Insight (CI) debugger, choosing the location of external events in the history being produced. So, for example, a user might chose the location for a context switch by setting a breakpoint, but then also specifying its context within the event history. It is easy to see why this is necessary if we consider a context switch taking place while a thread is executing a basic block within the body of a loop. In that case, the thread may have passed the \textit{location} of the breakpoint many times, but the event history \textit{context} distinguishes each time the thread passes that location from every other.

The user guiding the execution can thus provide CI with enough information to stop execution at the breakpoint \textit{within the desired execution history context}, rather than doing so every time and making the user decide if the context is correct or not. This is what developers typically do by personal inspection under current approaches, but in environments offering both incomplete information and incomplete control. CI can monitor the history produced by the executing program, checking it each time the breakpoint is passed. CI can also monitor the history \textit{as it is produced} to detect deviation from expected behavior, and thus avoid many causes of unbounded behavior. This checking of the produced history is easily implemented by attaching the procedure comparing the desired and produced histories to a breakpoint in the part of the computation recording the history. This is done with reasonable overhead by ensuring that the breakpoint is encountered only after a specific number of events have been generated, or a specific period of time has passed. However, certain classes of unbounded behavior, such as an unterminated wait on a non-responsive network connection, can only be resolved using timers.

It is also worth noting that it is possible for either a hu-
man or a model-driven agent to place the breakpoints. At each breakpoint the user can check whether the observed subsequence from the last breakpoint matches the specified sequence. If not, the difference will occur due to (1) an unanticipated or inadequately controlled concurrency influence, or (2) a mismatch between the program and the model of its execution. This solution may apply the Complete Event Sequence Recording and Event Sequence Execution patterns in its implementation.

3.1.4 Pattern P4: ESC

Name: Event Sequence Coverage (ESC)

Problem: Even with a set of hypotheses about event histories producing behaviors of interest, ensuring all possible influences on concurrency have been accounted for is much more difficult. Furthermore, enumerating all possible inter-leavings of application threads is a combinatorially hard problem.

Forces: Characterizing the behavior of concurrent software is hard (F1). Histories of interest may not occur during observed executions (F6). However, assurance requires covering the full range of relevant behaviors (F8).

Context: It is also possible to write event specifications that are not achievable under the semantics of the execution environment. We want to be able to compare the specified behaviors to the observed behaviors to detect discrepancies between the programming model and the execution environment, whatever their source.

Solution: Make the application semantics a first-class abstraction in the representation of coverage. We simply need a comparator that can determine whether or not two sequences are equivalent. For example, consider the discussion in Section 3.2. For a given comparator we can generate a candidate set of covering executions, as Figure 4 illustrates. We can either hypothesize an oracle, or refine successively through encountering examples as is done by classifiers in several AI approaches.

This solution also provides opportunity for heuristic search. Take for example the execution history of a single thread and all points at which it does a semaphore operation. If we place a context switch between each one and then do permutations for threads that are going to be interleaved, we will be able to generate and thus explore all histories of interest with respect to the semaphore operations.

The set of possible histories covers the set of behaviors that will be observed. This solution may use Event Sequence Synthesis to provide particular sequences to test specific hypotheses about the source of a problem, or as part of a test set to achieve certain degrees of coverage.

Consequences: There is a huge search space involved in exploring the space of histories that could be generated. It may be appropriate to apply model driven approaches to constrain the search space, and to improve the situation by organizing the exploration of the space.

Known Uses: Branch covering compilers that cover basic blocks, model checkers

3.2 Pattern Use Narrative

In this section we present a design narrative that is intended to illustrate a use of the patterns in the language at a greater level of detail than was appropriate when introducing the patterns. We use current work by two of the authors at the University of Kansas as an example. One reason for this is because while partial uses of the pattern language are common, complete uses are not. Another motivation is that the authors are most familiar with the relevant details of this project, which are required for an effective narrative. The project addresses the implementation of a program development and execution environment supporting reproducibility of concurrent computations, and contains three major components: a new GCC compiler feature supporting computation event recording, the BThreads user-level multi-threading library, and the Clever Insight debugger.
Supporting reproducibility within a programming environment requires a number of different capabilities, which fall into a natural implementation sequence within which each step steadily expands the types of computations for which reproducibility can be supported. We have implemented features that support reproducibility for compute-bound computations that may receive asynchronous signals. We will first discuss implementation details of our working prototypes and how they apply the patterns addressing event history recording, execution and synthesis. Then we will describe in Section 3.2.4 how these patterns and tools instantiating them could be used during the investigation of a classic concurrency control error scenario exhibiting intermittent irreproducible deadlock.

3.2.1 Complete Event Sequence Recording

The most fundamental capability required to support reproducibility is the ability to record the history of a computation as it executes. The most obvious events that must be recorded are from the execution of machine instructions by a thread. After some thought about the daunting volume of data that would be produced if we generated a record for each instruction, we realized that we could reduce the event history length without loss of information by recording the execution of each basic block (BB) of the program.

The term “basic block” comes from the technical area of language compilation and compiler implementation and refers to a segment of machine code produced by the compiler which is executed linearly without branching except perhaps at the end. Basic blocks are produced for some simple program statements in the source language, but many source program statements are translated into multiple basic blocks.

For example, a simple assignment statement $a = b$; in C would plausibly translate into the set of machine instructions for a target machine with a load-store architecture given in Figure 5(A). The CPU executes this sequence of two instructions without the possibility of changing the control flow by branching. More complex statements can also translate into a single basic block. For example, $a = b + c$ might translate into the basic block given in Figure 5(B). At this point it should be clear that a large expression involving only simple arithmetic operations on variables could translate into quite a large sequence of machine instructions that would still be a single basic block.

However, it should also be reasonably obvious that any source language statements that involve a change of control flow will be translated into multiple basic blocks. Such changes in the flow of control happen when subroutines are called, during conditional statement evaluation, and between successive iterations of a loop. The most obvious example is the conditional statement:

$$\text{if} \ (a == b)$$
$$a = a + 1;$$

This source statement will be translated into two basic blocks; one for the test of equality between $a$ and $b$, and one for the increment of $a$. The BB testing equality ends in a conditional jump instruction that targets either the basic block performing the increment of $a$ or the basic block beginning the next source statement following the conditional construct.

We can add a single event noting entry to the BB instead of an event for each BB instruction to the computation event history because of the linear character of the BB. In the vast majority of cases, a thread entering the beginning of a BB will leave it at the end without any complications. External events, such as context switches and signal delivery, can interrupt a thread’s execution of a basic block, but as these events note the thread’s current program counter value, we are able to generate the precise instruction-level event history easily from the BB-level history.

We have applied the Complete Event Sequence Recording pattern to multi-threaded computations that are compute bound, including cases involving asynchronous signal delivery. The resulting computation histories consist
of sequences of BB entry events punctuated by context switch and signal delivery events. The context switch and signal delivery events are generated by the BThreads library components responsible for user-level thread management.

The BB events are generated by instrumentation code inserted at the beginning of each basic block by a modified version of the GNU C compiler (GCC). The GCC modification for BB event generation uses an existing option that generates instrumented code for execution time profiling as a model. The profiling option adds code to the beginning of each BB, which increments a counter for that BB on entry. Post processing tools use the counter information to construct a statistical profile of how much of an application’s execution time is spent in various parts of the software. We used the basic framework of this option to add code to the beginning of each BB which emits an event history entry noting the unique label for the BB. The new GCC option also builds two mapping functions. The first takes a BB label as input and returns the address of its entry point. The other takes an address as input and returns the label of the BB within whose scope the address appears. These tables have been useful for ensuring the correctness and completeness of the recorded history, and will be useful when adding various features to the Clever Insight debugger to help users see what execution path through the code was taken by a particular instance of the computation.

Future extension of this application of the Complete Event Sequence Recording pattern will add the ability to handle a broader set of computations, including those performing read, write, and other system calls. In general, these events are slightly more difficult to record because we must add encapsulation layers for each system call to the BThreads library to enable us to generate the event and to capture any information returned to the computation by the call so that later replay of the history can provide the same information.

3.2.2 Event Sequence Execution

To support the Complete Event Sequence Recording we established by applying pattern P1, our application of the Event Sequence Execution (P2:ESE) pattern uses an enhanced source-level debugger to guide the computation’s execution as required to make it follow an input history, whether recorded or synthesized. Our approach to this problem was to use the Insight debugger as an implementation foundation, and to add a missing feature whose value was illustrated by a previous project, SmartGDB, supervised by one of the authors several years ago (1995-1998) [8]. The feature missing from Insight is the ability to attach an arbitrary procedure, written in a fully featured scripting language, to a break point for automatic execution when the breakpoint is encountered.

The SmartGDB project extended the GNU debugger GDB to address two drawbacks it had at that time; limited scripting capabilities, and limited support for controlling multi-threaded computations. The SmartGDB project integrated a Tcl/Tk scripting and GUI support framework with the command language of GDB, producing a hybrid scripting language within which it was possible to attach procedures containing both Tcl and GDB commands to breakpoints [8]. SmartGDB also contained an interface to a simple API in a threading package that explored the use of guided execution scenarios for multi-threaded programs by permitting the controller to force a context switch between threads at a breakpoint.

SmartGDB represents a partial application of P2:ESE pattern in the sense that it could guide the execution of a multi-threaded computation, forcing a context switch between threads at points selected by the user. However, it did not have the ability to read or follow an event history, even if we had been able to record or synthesize one. SmartGDB was targeted at several user-level thread packages and the Linux kernel threads package of the time, and was used as the basis for semester projects in the “EECS 448:Software Engineering I” class at the University of Kansas. The class experience concentrated on the wide variety of helpful debugger features that can be implemented using the ability to attach procedures to breakpoints.

Since then, the GDB development team has produced Insight, an extension to GDB that is similar to SmartGDB in supporting a command language including a superset of the GDB and Tcl commands, and providing access to Tk widgets from the debugger to support GUIs. GDB support for controlling multi-threaded computations has matured since SmartGDB was implemented. Insight also provides limited support for executing debugger code at a breakpoint, but it does not support attaching an arbitrary procedure written in the hybrid Tcl/GDB command language,
which proved such a powerful feature of SmartGDB [8].

We recently added this feature to Insight, calling the new version Clever-Insight (CI). With the ability to attach an arbitrary procedure to a breakpoint, it was comparatively simple to implement the ability to use a recorded computation event history to guide its execution under CI to reproduce the history for compute bound multi-threaded computations. CI reads the recorded event history and sets a breakpoint where the next context switch event occurs. Then, as the computation executes, CI checks the history being generated to see if it matches the context switching context. When it does, CI forces the specified context switch. If not, it executes the log of a concurrency scenario. The result is an accurate reproduction of the recorded concurrency scenario [13], and a full implementation of P2:ESE for compute-bound multi-threaded computations that can also receive signals.

3.2.3 Event Sequence Synthesis

Building on the previously established capability for Event Sequence Execution, our application of the Event Sequence Synthesis (P3:ESS) pattern relies on capabilities of the CI debugger for following an existing history, which include the ability to: (1) guide execution of multi-threaded computations, (2) read an input history and understand its elements, and (3) read the history emitted by the computation being guided. The application of the P3:ESS pattern uses these capabilities in service of the user who constructs the synthesized history as a series of segments punctuated by context switch and signal delivery events.

At each step in the construction process, the user in essence proposes the point in the future of the computation’s execution at which an optional event will occur. Context switches and signal delivery are examples of such optional events, and are the entire set of optional events for compute bound jobs, since they are the only sources of variation between executions for such computations. It is easy to see this if one imagines a compute bound computation with no input arguments, which reads no input information during execution, receives no signals, and contains a single thread. Such a computation produces the same history for every execution.

Synthesis of an execution history for a compute-bound multi-threaded computation thus requires the user to select the thread which will execute first, and then to select the point within the developing execution history of the initial thread at which a signal should be delivered, or at which a context switch to another thread should take place. In practical terms, this means the user must chose a place in the code at which to place a breakpoint. When the computation encounters that breakpoint during execution and returns control to the debugger, the user examines the context to decide it matches the desired context or not.

There are several points that should be considered here. First, the “synthesized” history is generated segment by segment by recording the history of the computation as the user guides its execution. Second, this process of guided execution provides a valuable check on the feasibility of a history because it is constructed by execution. In contrast, imagine that a user simply wrote out a candidate history by hand. It might or might not be a feasible history.

However, it is also true that a step by step check of such a hand-written history could be performed automatically by code written in the Clever Insight scripting language. The user would submit the candidate history, and CI would set a breakpoint at the location of the first optional event. CI would begin execution of the computation, and when the breakpoint was reached CI could check the first segment of the candidate history against that recorded for the current computation. If they matched, then CI would cause the optional event to occur, and repeat the process for the next segment of the candidate history. If the context of the recorded history at a breakpoint ever failed to match that predicted by the candidate history, then the candidate history would be flagged as being in error.

This is the essential core of the constructive method for synthesizing a candidate history. Various refinements are possible. First, CI need not wait until it reaches the breakpoint at the location of the next optional event in the candidate history. Instead, it could set a breakpoint in the code which records events, and attach a procedure checking the growing recorded history element by element against the candidate. Since doing this for every BB event would be tedious it would be useful to write the recording code cleverly, making it possible to set the breakpoint in a place that would only be reached every $N$ events. Another refinement would be to permit the user to describe the event history context in which the next
optional event should occur. CI could then monitor the growing history as it guides computation execution, stopping at a qualifying context to ask for approval from the user, or perhaps proceeding automatically.

We have implemented an initial prototype using these techniques for context switch events among compute bound threads. Similar techniques will be used as we extend this to signal delivery and I/O operations. Reproducing I/O operations will require the additional ability to supply the input data.

3.2.4 An Intermittent Deadlock Example

Having established capabilities for Complete Event Sequence Recording, Event Sequence Execution, and Event Sequence Synthesis, we now turn our attention to the question of Event Sequence Coverage. In this section we use process synchronization as a motivating example for the patterns and our current tools within which we have applied them. Consider two threads, $P$ and $Q$, running under the BThreads user-level multi-threaded execution environment, each with a critical section ($CS_P$ and $CS_Q$, respectively) that involves two separately managed shared data items. Entry to the critical sections therefore requires that two semaphores, $A$ and $B$, be acquired. In figure 6 we show code that attempts to solve this problem, but potentially suffers from deadlock. Note, however, that only a restricted set of computation event histories would display deadlock, and so this application might run for extended periods without deadlock. This is an example of an application which would be likely to demonstrate its flaw at best intermittently.

<table>
<thead>
<tr>
<th>Process $P$</th>
<th>Process $Q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td></td>
</tr>
<tr>
<td>P($A$)</td>
<td>P($B$)</td>
</tr>
<tr>
<td>1:</td>
<td></td>
</tr>
<tr>
<td>P($B$)</td>
<td>P($A$)</td>
</tr>
<tr>
<td>2:</td>
<td></td>
</tr>
<tr>
<td>$CS_P$</td>
<td>$CS_Q$</td>
</tr>
<tr>
<td>$V(B)$</td>
<td>$V(A)$</td>
</tr>
<tr>
<td>3:</td>
<td></td>
</tr>
<tr>
<td>$V(A)$</td>
<td>$V(B)$</td>
</tr>
<tr>
<td>4:</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6: Concurrent Threads with Intermittent Deadlock

Figure 6 divides the execution path of each thread into regions. Note that these regions may represent a very large number of machine instructions, but that the behavior of the processes is equivalent with respect to execution of the critical sections, regardless of where within these regions a context switch occurs. For example, a context switch away from, say, $P$ anywhere within region 2 implies that $P$ has acquired both semaphores. Hence, $Q$ is either executing in region 0 or region 4. In either case, $Q$ cannot execute in any other region until $P$ has a chance to run again.

Suppose that both $P$ and $Q$ execute the code in figure 6 exactly once. There are several possible execution histories of this code. We will identify executions by listing the series of context switches between $P$ and $Q$, and identifying the regions within which the context switches take place. For example, the execution $P_0 : Q_3 : P_1 : Q_4$ is the one in which process $P$ executes first, but a context switch occurs while it is still in region 0, then $Q$ executes until it reaches region 3, then $P$ executes until it reaches region 1, $Q$ executes until it reaches region 4, and $P$ then completes its execution.

Many arbitrary patterns of context switching are impossible due to the semantics of the semaphores. Furthermore, repeated context switches where a process remains in the same region can be collapsed for analysis purposes. That is, if the system switches from $P$ to $Q$ twice while $P$ is in region 0, then from the standpoint of the critical section executions, this execution is equivalent to one in which $P_0$ occurs only once. Hence, $P_0 : Q_1 : P_0 : Q_2 : P_0 : Q_3 : P_1 : Q_4$ is equivalent to $P_0 : Q_3 : P_1 : Q_4$. A little analysis shows that there are only a few possible equivalence classes of execution histories of this code with respect to its concurrency control. Assume, without loss of generality, that process $P$ executes first. Then the possibilities are:

- Executions $P_0 : Q_0 : P_1 : Q_1, P_0 : Q_1 : P_1,$ and $P_1 : Q_1$ are deadlocking executions.
- In executions $P_3 : Q_1 : P_4$ and $P_4 : P$ executes its critical section before $Q$.
- In executions $P_0 : Q_3 : P_1 : Q_4$ and $P_0 : Q_4, Q$ executes its critical section before $P$.

Therefore, although the number of possible executions may be extremely large when considering the sequence of
machine instructions to which these programs compile, all execution histories of the code belong to one of the equivalence classes listed above.

Now suppose that \( P \) and \( Q \) execute the code in figure 6 repeatedly. In this case, regions 0 and 4 are conceptually the same region, so we refer to both as region 0. Assume that when there is contention for a semaphore, one of the contending processes is arbitrarily chosen to acquire it. A similar analysis to the one above shows that the possible executions of this code are:

- Infinite executions in which \( P \) and \( Q \) take turns executing their critical sections. The \( P \) and \( Q \) executions are arbitrarily interleaved, and either \( P \) or \( Q \) may starve (i.e., not appear in the sequence at all).
- Finite prefixes of one of the above executions (including the empty prefix), followed by a deadlock state.

A developer of such an application who had some indication that there was a problem, would want to form an hypothesis about the cause as the first step in evaluating it, and if true, fixing the program by modifying it to eliminate the behavior hypothesized. He best way to discover such an hypothesis is to have a test case that relies produces the problem. If such a test case is available, then the pattern P1:CESR (via instrumenting GCC) could be used to record an event sequence for the execution, and the pattern P2:ESE (via Clever Insight) could be used to guide the execution of the computation while the developer examined the program context to determine the cause.

However, this example is more challenging than that because the problematic behavior, deadlock, depends on a possible but highly unlikely sequence of context switch events that are not under the program’s control. Specifically, a context switch must stop executing one thread, either \( P \) or \( Q \), while it is in region 1 and the other is in region 0 or 4. Then, the second thread must continue executing until it too reaches region 1. At that point each thread holds on semaphore and waits on the one held by the other thread, and deadlock has occurred. While possible, this situation is extremely unlikely because it depends on the coincidence of two essentially independent random variables: the location where the threads \( P \) and \( Q \) will stop when context switching to the other. In this case the probability that one of them is left by a context switch in a particular region is very small, and the probability that deadlock is produced is on the order of the product of very small probabilities.

Thus, it is far more likely that the developer of this application has only reports of intermittent freezing of the application requiring that it be terminated and restarted to indicate a problem, rather than a reproducible test case. In this situation the developer will have to form a hypothesis through another form of analysis. Typically this is done with a thorough examination of the code and a liberal dose of imagination. If a tool instantiating pattern P3:ESS (Clever Insight with the sequence construction feature) is available, then it is possible for the developer to construct a candidate sequence exhibiting the behavior. Alternatively, another tool might use a model based method to construct a candidate computation history.

In any case, supplied with a synthesized history that purports to produce the problematic behavior, the developer can use a tool instantiating pattern P2:ESE to guide the execution of the computation according to the synthesized history to evaluate the validity of the hypothesis in the context of the application source code, and the program state as it executes.

While this scenario shows that such problems with concurrency control remain subtle puzzles that are difficult to solve, it should also be obvious that a tool set and programming environment instantiating the patterns in the language provide significantly improved support for the developer in forming and testing hypotheses in the process of diagnosing and fixing the error in concurrency control.

It is worth considering the pattern P4:ESC, Event Sequence Coverage, in the context of this example. Recall that a fairly simple analysis of this computation using two threads was able to divide the set of all event histories into a set of equivalence classes, with respect to the concurrency control and context switch events. A test suite that is complete with respect to the concurrency control analysis will cause the computation to produce a set of event sequences containing at least one sequence from each equivalence class. A minimum size complete test suite will cause the computation to produce exactly one event sequence from each equivalence class.

Finally, it should be obvious that the failure of most program development and execution environments to instantiate patterns P1:CESR, P2:ESE, and P3:ESS make it
impossible for them to instantiate P4:ESC. Use of the pattern language described in this paper to improve the feature set of future programming environments thus holds the promise of significant improvement over current practice.

4 Related Work

Many researchers have investigated using source-code transformations to support reproducibility. This work includes support for Ada [17], Concurrent Pascal [9], and SR [14]. The transformational approach has certain limits, such as the need for a separately implemented transformation is required for each synchronization structure, and limits on completeness of a transformation with respect to representing all possible interleavings of concurrent threads or forms of concurrency outside threading. Despite these limits, we view this pattern language as relevant to improving predictability in source transformations as well as in system infrastructure.

Albertsson, et al., have produced a whole machine simulator for Linux, and simulation-based debuggers for soft real-time applications [2, 1]. Executions of the simulated machine are deterministic, since deterministic timing models are used. This means that executions are reproducible, but it also means that a given concurrent program will execute in only one way. Application of this pattern language in its entirety would allow exploration of the space of all possible executions of a given program.

The Rivet Java Virtual Machine contains a replay mechanism that exemplifies the Complete Event Sequence Recording design pattern presented here [3]. However, it increments a counter on every JVM instruction, which can produce data at levels and rates that would overload system resources. Applying the rest of our pattern language to this design could result in emitting a concurrency event only once per basic block or gathering aggregate information about the instruction stream in-place, instead of generating an event-per-instruction stream.

PVM has been extended with a series of increasingly capable debuggers that support distributed debugging [4, 5, 7]. These debuggers require the distributed application to log information as it runs, and this information is then collected at a central site. Debugging interfaces in the application software allow the debugger to control their actions, forcing the distributed computation to back up and replay actions in some cases. While a great deal of work has gone into compressing and optimizing the data collection process to reduce the impact on the running program, that impact is still perceptible. Our pattern language is potentially able to generate environments running on single machines, that can simulate execution of distributed computations with imperceptible impact on the running programs.

There is also a large body of related work in the debugging literature. Summaries of existing research are given for debugging of parallel systems in [10], and for debuggers based on execution replay in [6].

This pattern language is applicable to testing techniques explored in the literature, such as deterministic testing, prefix-based testing [16], and reachability testing [11]. The patterns presented here are intended to allow those techniques to be implemented more simply and effectively.

5 Conclusions

We have presented a pattern language consisting of four design patterns, presented in Section 3 for achieving reproducible behavior of concurrently executing computations. These patterns address the specific design forces described in Section 2, and a generative design using these patterns is presented in Section 3.2.

References


