Depiction and Playout of Multi-threaded Program Executions

Abhik Roychoudhury
School of Computing
National University of Singapore
abhik@comp.nus.edu.sg

Abstract

Execution of a shared memory multi-threaded program is non-deterministic even for a fixed input. Consequently a limited amount of the program behavior should be traced and recorded during run-time. However, if the tracing overheads are too high, we have the risk of slowing down the program considerably and even distorting the program behavior. In this paper, we propose to collect and store only the synchronization dependences during run-time. These dependences are visualized as a Message Sequence Chart (MSC). We do not record the data dependences across threads resulting from unsynchronized reads and writes of a shared variable. Instead all possible orderings of unsynchronized reads/writes are analyzed post-mortem. To describe all these behaviors, we use an important extension of Message Sequence Charts called Live Sequence Charts (LSC). Our MSC/LSC based description of a multi-threaded program execution can be simulated in an automated manner. This can help in understanding program behavior.

1. Practical Challenges in Depiction

We first describe some of the difficulties in visually capturing a shared memory multi-threaded program execution. For shared memory multi-threaded programs, a distinction is usually made regarding whether the program is run on a uni-processor or multi-processor platform. In this paper, we simply identify the information in a multi-threaded program execution which need to be captured (irrespective of whether the program is run as a parallel or concurrent program). Clearly, in a multi-threaded program execution, we need to represent: (a) synchronization pattern (order of lock/unlock operations), and (b) data dependences for shared variables (for each read operation of a shared memory location, the write operation whose result is returned by the read operation)

The fact that these information need to be tracked in order to represent a multi-threaded program execution is well-known (see for example [1, 4, 9]). Unfortunately, shared variable accesses are not easy to track during program execution. The order in which updates to a shared location by different threads is committed is impossible to observe at the program level. In particular, for a shared memory parallel program run on a bus-based multiprocessor system, this would require observing and recording every bus transaction leading to a flow of data from processor cache to main memory. This is expensive and involves very low-level hooks [4, 10].

In this paper, we suggest a different approach. We try to record only the synchronization information during program execution. This is recorded as a partially ordered sequence of operations, or a Message Sequence Chart (MSC) [2, 12]. If we want to use the MSC for debugging, then we execute and analyze it post-mortem to find the exact execution trace (*i.e.* the shared variable data dependences) that caused the bug in question. For this purpose we do not need to record/trace the global order of shared variable reads/writes. Instead we only need to know the list of shared variables/objects (which may be user-provided) ; accesses to these variables are recorded locally within a thread during run-time.

Capturing Synchronization Patterns Let us now describe how a MSC can be used to capture the synchronization pattern of a trace. To represent the execution of a multi-threaded program $P_1 \parallel \ldots \parallel P_n$ we depict each thread P_i as a vertical line. Furthermore, we represent an additional process M denoting a single shared store containing *all* the lock variables. In general there can be many locks in a realistic program, each guarding different structures/objects. In this paper, we represent all of them by a single memory process M. It is also possible to consider a *minor* variation of this scheme where each lock variable is drawn as a different process. In fact, for concurrent programs (where one thread at a time is chosen by the scheduler), a thread being scheduled-in and scheduled-out can also being depicted as the acquire and release of a special lock.

Each unlock operation in thread P_i over lock variable l is shown as a send event $P_i!M, l$. Similarly, a lock operation in thread P_i over lock variable l is a receive event $P_i?M, l$. In other words, send/receive events are used to depict hand-over of locks. All other computations, including read/write of shared variables are considered as internal events.

An example MSC depicting the synchronization pattern

```
lock l, unlock l (P1)
lock l, unlock l (P2)
lock l, unlock l (P3)
lock l           (P1)
```

is shown in Figure 1. In the above execution, `l` is a lock variable. We also show two sample read/write operations which are in "conflict", that is, they are not required to be ordered by the lock/unlock operations. In other words, X is a shared variable and these two operations are unsynchronized shared variable updates. They constitute a data race; the value returned of the read operation in P_1 depends on whether the write operation in P_2 has been completed.[1]

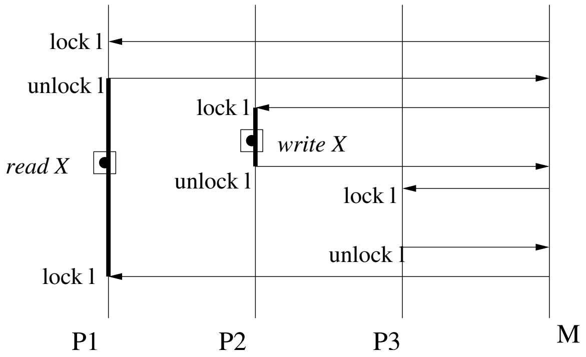


Figure 1. Depicting synchronization patterns

Capturing data dependences As is clear by now, the MSC description does not capture data dependences in a multi-threaded program execution trace. In particular, data dependences of unsynchronized shared variable accesses (data races) are not captured. Since data races are the source of many bugs in multi-threaded programs, this is a cause of serious concern ! Ideally, one would like the following solution. The possible flows of data to and from the memory via reads and writes should be depicted in the MSC. This also allows for post-mortem analysis of the data dependences. However, a single MSC should represent several traces by capturing the non-determinism of a shared variable read (a shared variable read can return the value of different writes in different traces).

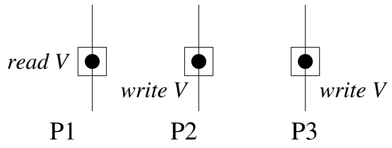
1 Note that if the program is run on a multi-processor platform then P_1 can execute even while P_2 has acquired the lock, and is executing.

Example Let us now examine whether the MSC visual formalism is powerful enough to capture this non-determinism of a shared variable read. Consider three threads $P1$, $P2$ and $P3$ executing the following program; `V` is a shared variable and all the shared variable accesses are unsynchronized.

$P1$	$P2$	$P3$
`read V` $\|$	`write V` $\|$	`write V`

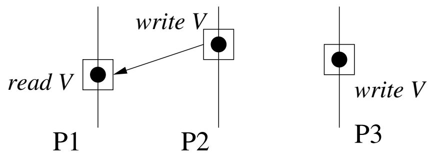
Figure 2. Example program fragment

Since there are no synchronization operations, the following MSC trivially captures all the execution traces, that is, all the orderings of the three operations.



This diagram simply shows the reads/writes as local operations and does not capture the fact that these operations involve interaction with other threads (via shared memory). However, this defeats the entire purpose of using MSCs to depict scenarios in concurrent/distributed systems. MSCs provide a clear graphical description of communication in specific runs of such systems. Denoting the shared variable accesses as local operations amounts to not representing inter-process communication.

Shared Variable Interaction Diagrams (SVID) Whether we can capture inter-process interactions in a shared memory concurrent/distributed system via MSCs is an important and intriguing question. Since MSCs have been primarily developed for visualizing message-passing style of concurrency, a valid question is whether they also capture interactions in shared memory concurrency. A recent work by Alur and Grosu [1] introduces a modification of the MSC formalism (called Shared Variable Interaction Diagrams or SVID) for this purpose. The specification style of SVIDs can be explained as follows. In the above program, a data dependence `write V` $(P2) \rightarrow$ `read V` $(P1)$ is represented by the following diagram. This shows explicitly that `read V` returns the value written by $P2$.



Clearly, SVIDs also represent multiple traces. For example, the above diagram captures both the traces

`write V`($P2$)	`write V`($P3$)
`read V`($P1$)	`read V`($P1$)
`write V`($P3$)	`write V`($P2$)
(a)	(b)

Figure 3. Two traces causing different results

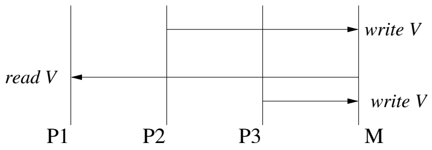
`write V`($P2$), `read V`($P1$), `write V`($P3$)
`write V`($P3$), `write V`($P2$), `read V`($P1$)

However, since SVIDs explicitly record data dependences, they cannot capture the non-determinism of a read operation. That is, they cannot represent multiple traces which differ from each other in the values returned by constituent read operation(s). An example of such a pair of traces appears in Figure 3. It is not clear how SVIDs can be extended to capture this non-determinism.

How can we capture non-determinism of a read ? Let us recall our original goal. We want to represent only the synchronization information in our depiction of a multi-threaded program trace. However, in order to trace a bug, we want to analyze the depicted program trace post-mortem to find the data dependences relevant to the bug. The question is: *Can we use/adapt MSCs to represent several multi-threaded program traces, which have the same synchronization pattern but different data dependences ?*

Suppose we want to represent the two traces in Figure 3 as a MSC (these are traces of the program fragment in Figure 2). The two traces differ in the value returned by `read V`. Note that there are other traces in this program, but these two traces are sufficient to show the problem. Clearly, if we represent data dependences by drawing arrows from a write operation to a read operation (in the style of Alur and Grosu [1]), we cannot represent these two traces in a single MSC. This would require two incoming arrows into `read V` which is ambiguous according to [1].

Another natural specification would be to *explicitly* capture the flow of data by representing a shared variable write operation by thread P_i as an arrow between process P_i and the *M* process. Similarly, a shared variable read by thread P_i is represented as an arrow from the *M* process to process P_i. In this style, the trace in Figure 3(a) is depicted as:



Even in this approach, we cannot depict both the traces of Figure 3 as a single MSC. Note that the events in a particular process of a MSC are totally ordered. Thus, the two `write V` operations in the traces of Figure 3 will have to be ordered in process M. This effectively defines the value returned by a `read V` operation (value of the write that occurs later in the *M* process of the MSC). As a side remark, we note that simple modifications of the depiction scheme such as representing each shared variable as a separate process in the MSC (instead of a single shared memory process) will not solve this problem. In the example traces of Figure 3, both writes refer to the same shared location.

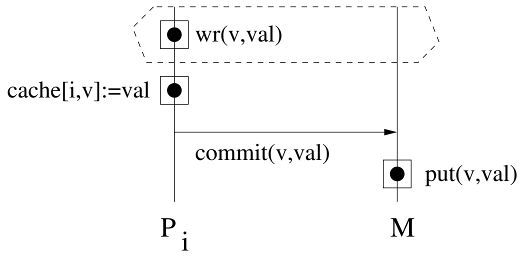


Figure 4. Univ. chart for shared variable write

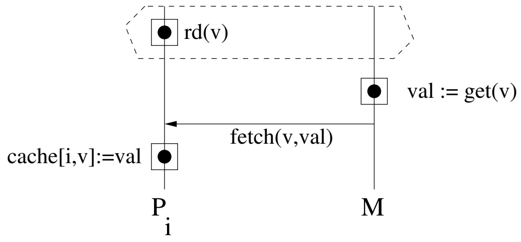


Figure 5. Univ. chart for shared variable read

2. Using Universal Charts

We now use Live Sequence Charts [3] to pictorially represent a multi-threaded program execution. The primary benefit of using LSCs is that we can use *universal charts*, charts which denote safety properties. These charts have two parts: a pre-chart and a body chart. If the pre-chart is satisfied, the body chart *must* eventually be satisfied.

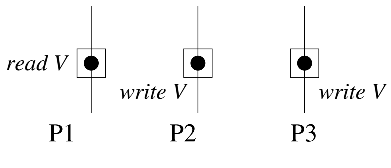
The synchronization pattern of a multi-threaded program can be depicted using a MSC as shown in Figure 1. As far as the shared variable data accesses are concerned, we do not want our depiction to impose orderings on unsynchronized shared variable accesses in an execution. Thus, we represent a shared variable read/write by thread i as an internal action of thread i. As mentioned in Section 1, this however ignores the interaction between different threads of a multi-threaded program (via the shared memory).

To avoid this problem, we augment our specification with two universal charts in the style of LSCs. One of the charts is activated when any shared variable write takes place. The other one is activated in the case of a shared variable read. The charts are shown in Figures 4 and 5. As per LSC conventions, the pre-chart of any chart is shown within dashed lines. Each chart involves two processes P_i and M. Note that P_i is a symbolic process instance denoting thread i; M denotes the shared memory process. Each thread P_i is assumed to have a local copy of the shared variables. Thus, the local copy of shared variable v for thread i is $cache[i, v]$. We assume two internal primitive actions for the shared memory process M which lets it read/write the master copy of the shared variables. These primitive actions are `get` and `put`. Thus `put(v,val)` write the value `val` to the master copy of variable `v`. Similarly `get(v)` returns the value of the master copy of shared variable `v`. These primitives are not central to our specification. We only need them to abstract the implementation details of the shared memory process.

In the chart for shared variable write, the pre-chart involves a write event to shared variable `v` by thread P_i. This write event is invoked by some statement in the multi-threaded program; it activates the main body of the chart. Subsequently, the local copy ($cache[i, v]$) is updated. This is followed by a commit to the main memory, and the updating of the master-copy. Similarly, in the chart for shared variable reads, the pre-chart is a read event for shared variable `v` by the program running in thread i. The main body of this chart involves fetching the value of the master copy from shared memory process M.

Let us now study the important features of the universal charts. This will clarify the role of these charts.

- We have de-linked the start and finish of a shared variable read/write. In fact, shared variable accesses involve communication between threads and shared memory. Shared variable writes are started by some thread P_i and finished by the shared memory M. This allows us to capture the non-determinism of a shared variable read. For example, let us again consider the two traces shown in Figure 3. They are traces of the program in Figure 2. Both these traces will be captured if we represent the execution as the following MSC, along with the universal charts of Figures 4 and 5.



- Computation events in any thread P_i always access local copies of the shared variables. These events never access the master copy (which is local to the shared memory process M).

- By definition, any shared variable read/write activates the corresponding universal chart. Thus, two writes to a shared variable v will activate two live copies of the chart for writing. Both these copies are guaranteed to eventually run to completion, that is, both the writes will be committed. This is required to ensure that no writes are lost. However, their relative order of completion is not fixed a-priori. Different orderings of the commits can be obtained by executing the charts post-mortem in a simulation engine (such as the LSC play-engine [5]).

- Note that in the two charts `v` and `val` are symbolic variables. For any shared variable read/write they will get instantiated by the corresponding variable name and value. Furthermore process P_i is a symbolic instance. Both these features are supported in the current LSC specification language [8].

- The keen reader may notice that we commit *every* shared variable write, whereas in realistic implementations this is not the case. However, our aim is to represent all possible behaviors resulting from writes to shared variables by different threads (which may not be immediately observable in all other threads). This is captured since whenever the pre-chart is activated (for the charts in Figures 4 and 5) in thread i, the main body of the chart need not immediately follow. Clearly, certain events in thread j ($i \neq j$) can take place between one of the pre-charts of Figures 4, 5 involving thread i and the main body of the corresponding universal chart.

We remark that our depiction scheme draws important lessons from how shared variable reads/writes can be implemented in uni- and multi-processor platforms. This is to capture all possible behaviors in a multi-threaded program execution. However, our depiction does not describe the actual implementation of shared memory accesses.

3. A Depiction Scheme using LSCs

In this section, we present our scheme for visualizing multi-threaded program traces in an MSC-like format. Our scheme supports lock/unlock, wait/notify as well as fork/join. Thus, it can be used to describe the execution profile of any multi-threaded Java program. We describe the synchronization information in a multi-threaded program trace as an MSC. If there are n threads in the multi-threaded program, the processes in the MSC are:

- $P_1, \ldots, P_n$ denoting the n threads
- a shared memory process M

The events in P_i as well as M are depicted as follows.

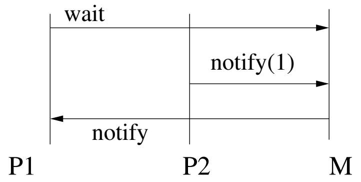


Figure 6. Waiting and Notification

Lock and Unlock Lock and unlock events in thread i are denoted by exchange of messages between P_i and M. In fact, lock corresponds to a message from M to P_i while unlock corresponds to a message from P_i to M. Lock/unlock being reading/writing of master copies of shared lock variables, this forms a natural specification of lock and unlock.

Waiting/Notification of threads We can also depict waiting and notification of threads in a trace as follows. Execution of `wait` by thread i is denoted by a message from process P_i to the shared memory process M. Notification of thread j by thread i can be depicted as a message from P_i to shared memory M followed by a message from M to P_j. A wait issued by thread 1 followed by notification of thread 1 by thread 2 is depicted as in Figure 6.

Computation Each thread P_i has local variables corresponding to the local copies of the shared variables. The local copy of shared variable v in thread i is denoted as $cache[i, v]$. Internal computations in thread i are denoted as internal actions of process P_i involving local copies.

Read/Write of Shared Variables The read/write of a shared variable v in thread i is a internal event which triggers one of the universal LSCs of Figures 4 and 5 (depending on whether it is a read or write event). The execution of the universal chart results in a transfer of data between local and master copies. Consequently, the local copy $cache[i, v]$ is also updated.

Fork/Join of threads Finally, we need to depict forking and joining of threads in a multi-threaded program trace. Forking of thread j by thread i is shown by an arrow from P_i to P_j (denoting the start event of thread j). Joining of thread j into thread i is shown by an arrow from P_j to P_i (denoting the finish event of thread j).

4. Post-mortem Playout

We now discuss the post-mortem simulation and analysis of the collected execution profile. During execution, we only track the input of each thread as well as the synchronization operations executed. In the post-mortem phase, we use this information to re-generate the portion of code executed in each thread. During this re-generation phase, whenever we encounter a shared variable read or write operation, we start a live copy of the corresponding universal chart. Any live copy generated in this way is terminated only when the corresponding shared variable access is completed. In the case of a shared variable write, this means the committing of the write into the shared memory; in the case of a shared variable read, this means the fetching of the value from the shared memory.

Semantics of lock/unlock Note that any live copy of a universal chart generated by a shared variable read/write ensures that the read/write operation will be eventually completed (*i.e.*, the body of the universal chart will be executed). However, to respect the semantics of synchronization operations like lock/unlock we need to ensure that all operations issued by a thread between a lock and an unlock must be executed as a critical section during the post-mortem playout. In other words, operations issued by any thread between a lock l and the corresponding unlock u should be completed before u. This is easily ensured by enforcing the following rule during post-mortem playout: a thread P_i can execute an unlock operation only when live copies of all universal charts started by P_i have terminated.

Memory Consistency Models A multi-threaded program can be executed over uni-processor as well as multi-processor platforms. For uni-processor platforms, the operations in each thread appear to execute in program order. Thus, the model of execution supported is Sequential Consistency [7]. Sequential Consistency demands that (a) operations across threads can be arbitrarily interleaved, and (b) operations in each thread proceed in program order. This is not the case in multi-processor platforms which support more relaxed memory consistency models. For example, SUN SPARC can support the Total Store Order (TSO) which allows committing of writes in a thread to be delayed; this causes reads in a thread to bypass preceding write operations. For programs which are not properly synchronized (every shared variable read/write is not executed in a critical section), multi-processor platforms with relaxed consistency models allow more behaviors. These behaviors are also allowed in our execution profile, since we allow re-ordering of operations within a thread.

Our universal charts for shared variable accesses de-link the start and finish of a shared variable read/write. This was done to support any order of execution for unsynchronized shared variable accesses in two threads. By de-linking the start and finish of shared variable accesses, we also allow shared variable reads/writes to be executed out of order *within a thread*. This corresponds to the additional behaviors allowed by the relaxed multiprocessor memory consistency models. Consequently our post-mortem playout of execution traces can reveal those aberrant behaviors of multi-threaded programs which are displayed only on certain multi-processor platforms (and result from the relaxations allowed by the underlying multiprocessor consistency model). Real-life examples of such aberrant behaviors include a possible garbage object being returned in the well-known Double Checked Locking idiom which performs lazy initialization of a singleton Java object [11].

Currently, the only ordering restriction in our playout mechanism (all initiated shared variable accesses are completed before unlock) aims to preserve the semantics of lock/unlock. Of course we can place more severe restrictions such as preserving Sequential Consistency, where we do not consider any behaviors involving operation reordering in a thread. For post-mortem playout of multi-threaded Java programs, another possibility is to enforce the Java Memory Model which dictates the set of all possible behaviors of any multi-threaded Java program on any implementation platform [6].

5. Discussion

In this paper, we have studied the problem of collection, depiction and post-mortem simulation/analysis of multi-threaded program executions. Based on practical difficulties in tracking shared variable updates during run-time, we propose to track and collect only synchronization operations. Inter-thread communication via unsynchronized shared variable updates (and their possible relative orderings) are analyzed post-mortem. This is made possible by (a) depicting the collected synchronization pattern as a Message Sequence Chart, and (b) employing Live Sequence Charts to capture data dependences due to shared variable updates. The MSC/LSC based specification constitutes a partial description of the execution trace. All traces *respecting this specification* are played out in the post-mortem analysis (in a manner similar to state space exploration).

The work in this paper describes a novel use of Live Sequence Charts for depicting shared memory multi-threaded program behavior. To the best of our knowledge, there has been little systematic study to describe shared memory program behavior using the message passing style of MSCs. The work of [1] presents a solution to this problem, but cannot handle program traces with different data dependences (and hence different "results"). In this paper, we use the universal charts of the LSC formalism to address this limitation. Due to the executable nature of LSC specifications, our depiction is also amenable to post-mortem simulation and analysis.

6. Acknowledgments

This work was partially supported by National University of Singapore Research Project R-252-000-095-112.

References

[1] R. Alur and R. Grosu. Shared variable interaction diagrams. In *International Conference on Automated Software Engineering (ASE)*, 2001.

[2] R. Alur, G. Holzmann, and D. Peled. An analyzer for Message Sequence Charts. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS), LNCS 1055*, 1996.

[3] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1), 2001.

[4] A. Goel, A. Roychoudhury, and T. Mitra. Compactly representing parallel program executions. In *ACM International Conference on Principles and Practice of Parallel Programming (PPoPP)*, 2003.

[5] D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart play-out of behavioral requirements. In *International Conference on Formal Methods in Computer Aided Design (FMCAD)*, 2002.

[6] Java Memory Model and Thread Specification revision, 2001. Available from `http://jcp.org/jsr/detail/133.jsp`.

[7] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9), 1979.

[8] R. Marelly, D. Harel, and H. Kugler. Multiple instances and symbolic variables in executable sequence charts. In *Annual. ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2002.

[9] R. Netzer and B. Miller. Improving the accuracy of data race detection. In *ACM International Conference on Principles and Practice of Parallel Programming (PPoPP)*, 1991.

[10] M. Ronsse and K. de Bosschere. Recplay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17(2), 1999.

[11] D. Schmidt and T. Harrison. Double-checked locking: An optimization pattern for efficiently initializing and accessing thread-safe objects. In *3rd Annual Pattern Languages of Program Design conference*, 1996.

[12] Z.120. Message Sequence Charts (MSC'96), 1996.